US 20120179760A1

(19) **United States**

(12) **Patent Application Publication**

**Blocksome et al.**

(10) **Pub. No.: US 2012/0179760 A1**

(43) **Pub. Date: Jul. 12, 2012**

(54) **COMPLETION PROCESSING FOR DATA COMMUNICATIONS INSTRUCTIONS**

(75) Inventors: **Michael A. Blocksome**, Rochester, MN (US); **Sameer Kumar**, White Plains, NY (US); **Jeffrey J. Parker**, Rochester, MN (US)

(73) Assignee: **INTERNATIONAL BUSINESS MACHINES CORPORATION**, Armonk, NY (US)

(21) Appl. No.: **12/985,651**

(22) Filed: **Jan. 6, 2011**

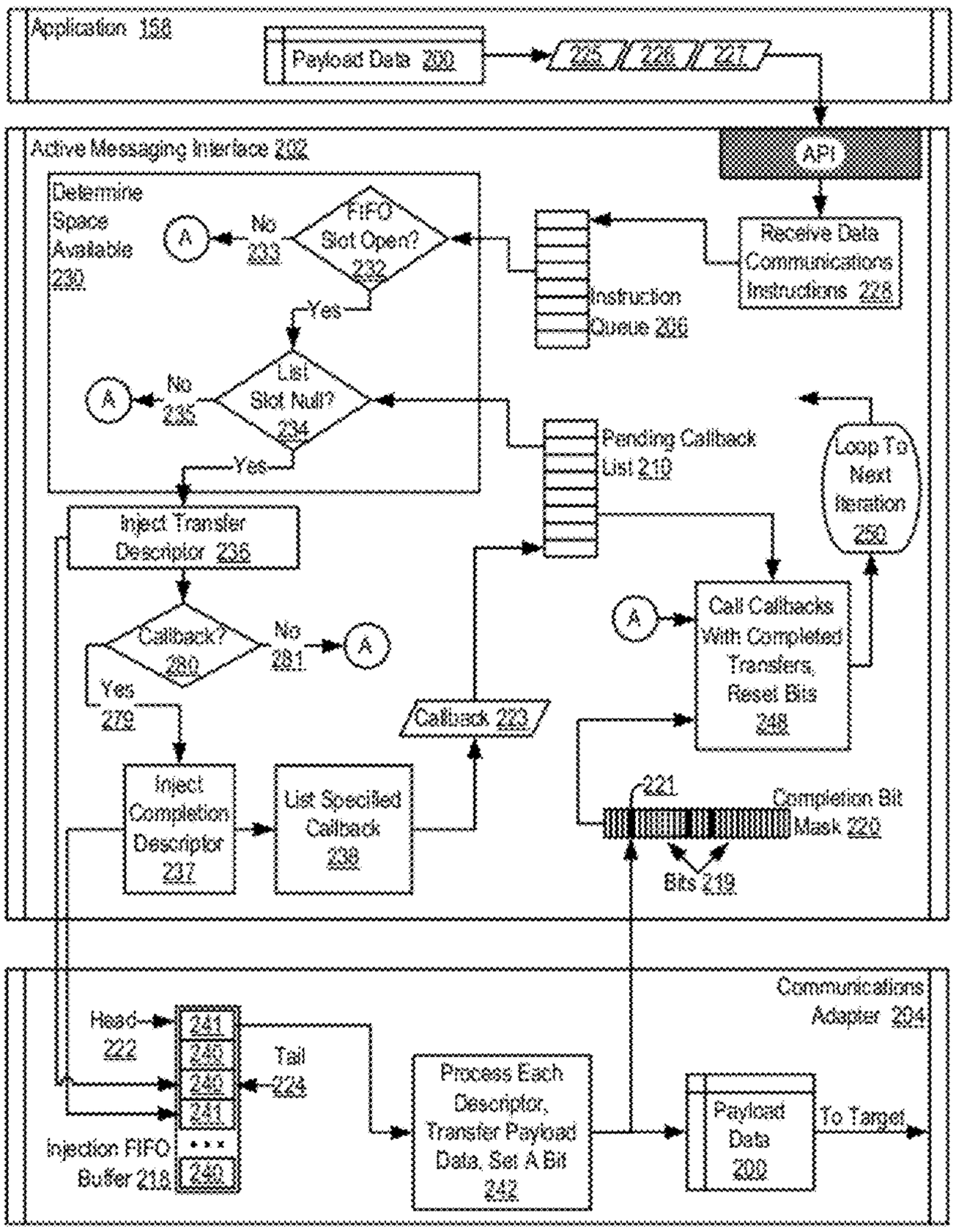
**Publication Classification**

(51) **Int. Cl.**
***G06F 15/16*** (2006.01)

(52) **U.S. Cl.** ..... **709/206**

(57) **ABSTRACT**

Completion processing of data communications instructions in a distributed computing environment with computers coupled for data communications through communications adapters and an active messaging interface ('AMI'), injecting for data communications instructions into slots in an injection FIFO buffer a transfer descriptor, at least some of the instructions specifying callback functions; injecting a completion descriptor for each instruction that specifies a callback function into an injection FIFO buffer slot having a corresponding slot in a pending callback list; listing in the pending callback list callback functions specified by data communications instructions; processing each descriptor in the injection FIFO buffer, setting a bit in a completion bit mask corresponding to the slot in the FIFO where the completion descriptor was injected; and calling by the AMI any callback functions in the pending callback list as indicated by set bits in the completion bit mask.



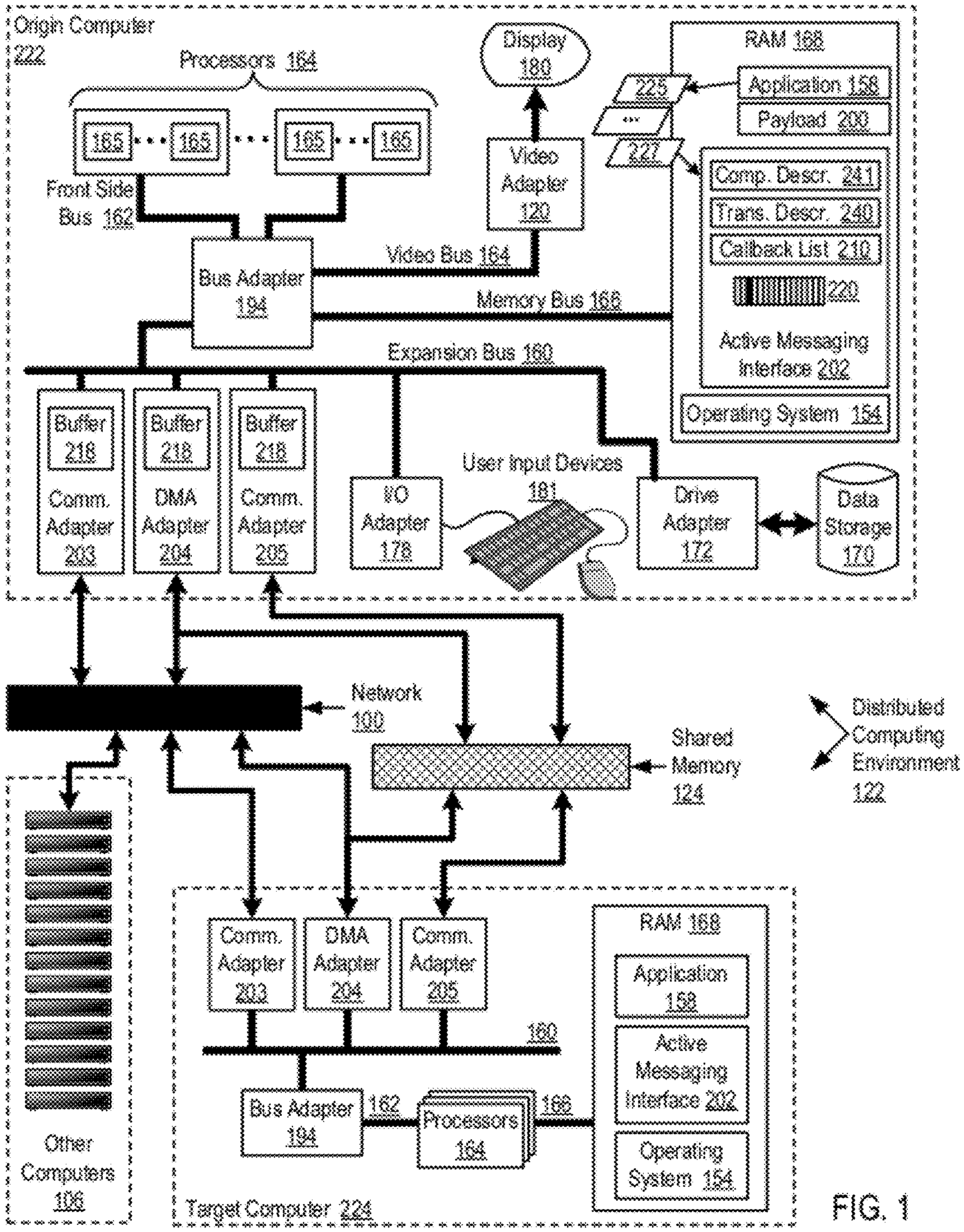

Origin Computer 222
Processors 164
165
165
165
165
Front Side Bus 162
Display 180
Video Adapter 120
Video Bus 184
Bus Adapter 194
Memory Bus 166
Expansion Bus 160
Buffer 218
Comm. Adapter 203
Buffer 218
DMA Adapter 204
Buffer 218
Comm. Adapter 205
I/O Adapter 178
User Input Devices 181
Drive Adapter 172
Data Storage 170
RAM 168
225
...
227
Application 158
Payload 200
Comp. Descr. 241
Trans. Descr. 240
Callback List 210
220
Active Messaging Interface 202
Operating System 154
Network 100
Shared Memory 124
Distributed Computing Environment 122
Other Computers 106
Comm. Adapter 203
DMA Adapter 204
Comm. Adapter 205
160
Bus Adapter 194
162
Processors 164
166
RAM 168
Application 158
Active Messaging Interface 202
Operating System 154
Target Computer 224

FIG. 1

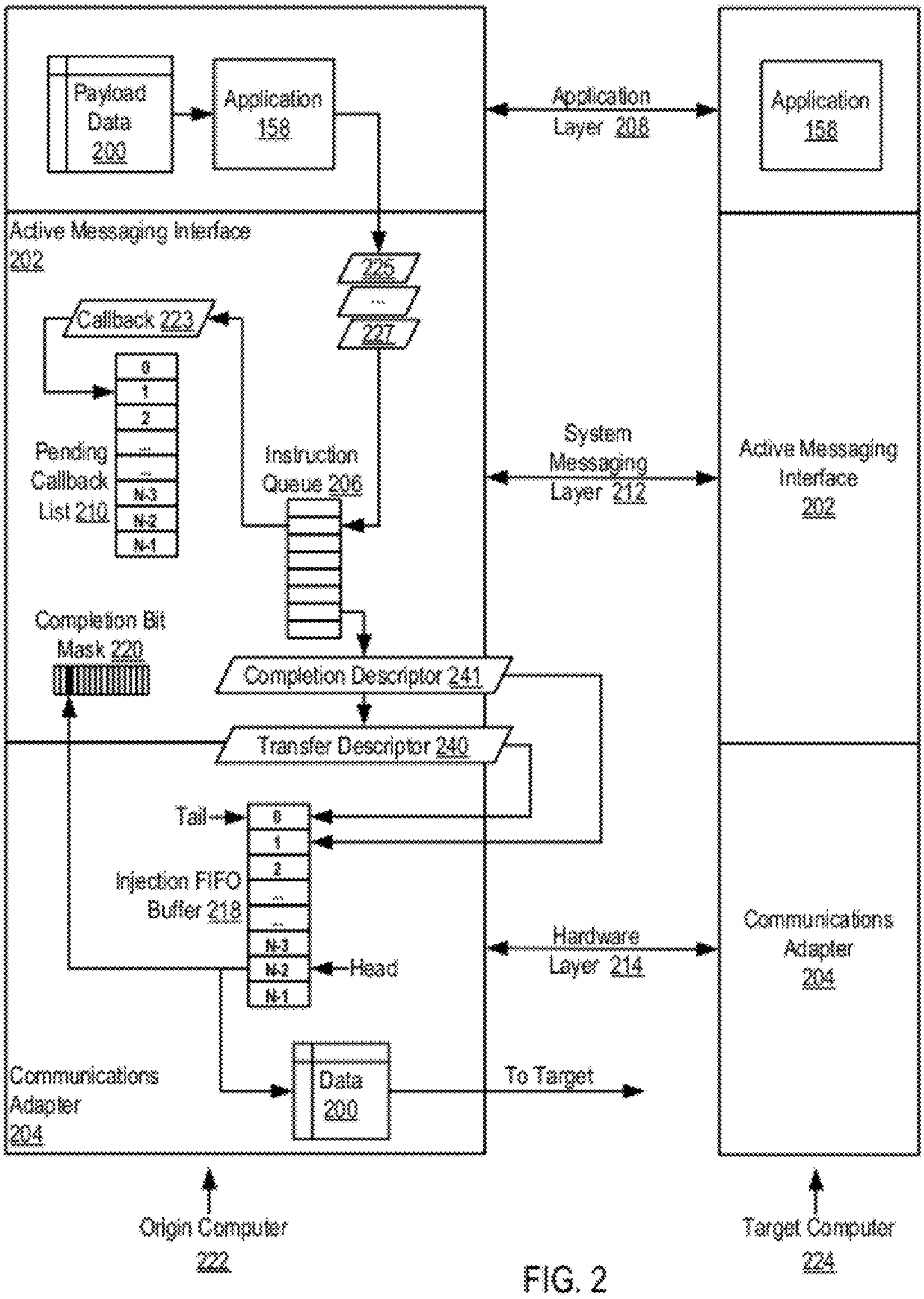

Payload Data 200
Application 158
Application Layer 208
Application 158
Active Messaging Interface 202
225
...
227
Callback 223
0
1
2
...
...
N-3
N-2
N-1
Pending Callback List 210
Instruction Queue 206
System Messaging Layer 212
Active Messaging Interface 202
Completion Bit Mask 220
Completion Descriptor 241
Transfer Descriptor 240
Tail
0
1
2
...
...
N-3
N-2
N-1
Head
Injection FIFO Buffer 218
Hardware Layer 214
Communications Adapter 204
Communications Adapter 204
Data 200
To Target
Origin Computer 222
Target Computer 224

FIG. 2

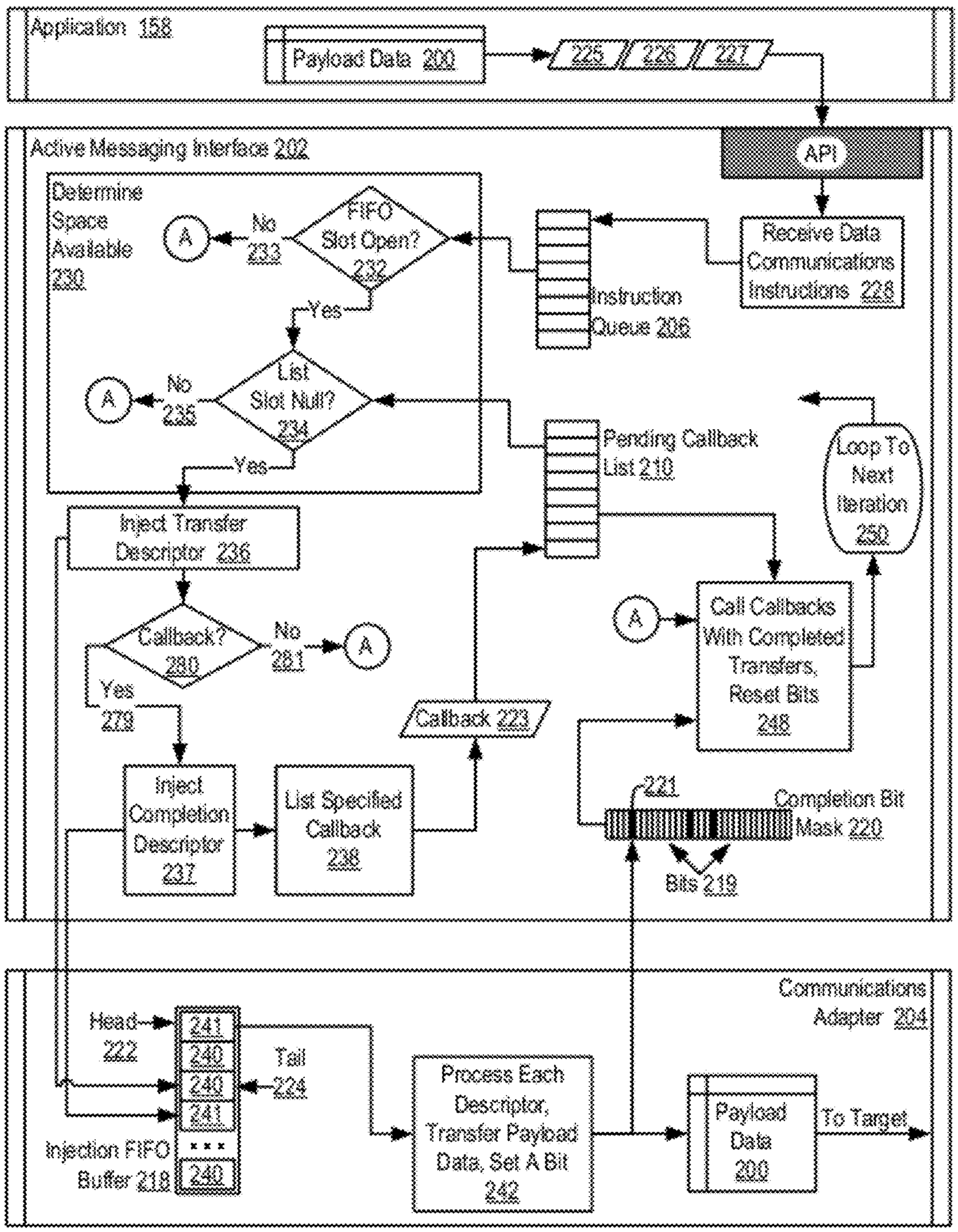

Application 158
Payload Data 200
225
226
227
Active Messaging Interface 202
API
Determine Space Available 230
FIFO Slot Open? 232
No 233
A
Yes
List Slot Null? 234
No 235
A
Yes
Instruction Queue 206
Receive Data Communications Instructions 228
Pending Callback List 210
Loop To Next Iteration 250
Inject Transfer Descriptor 236
Callback? 280
No 281
A
Yes 279
Callback 223
A
Call Callbacks With Completed Transfers, Reset Bits 248
Inject Completion Descriptor 237
List Specified Callback 238
221
Completion Bit Mask 220
Bits 219
Communications Adapter 204
Head 222
241
240
240
241
• • •
240
Injection FIFO Buffer 218
Tail 224
Process Each Descriptor, Transfer Payload Data, Set A Bit 242
Payload Data 200
To Target

FIG. 3

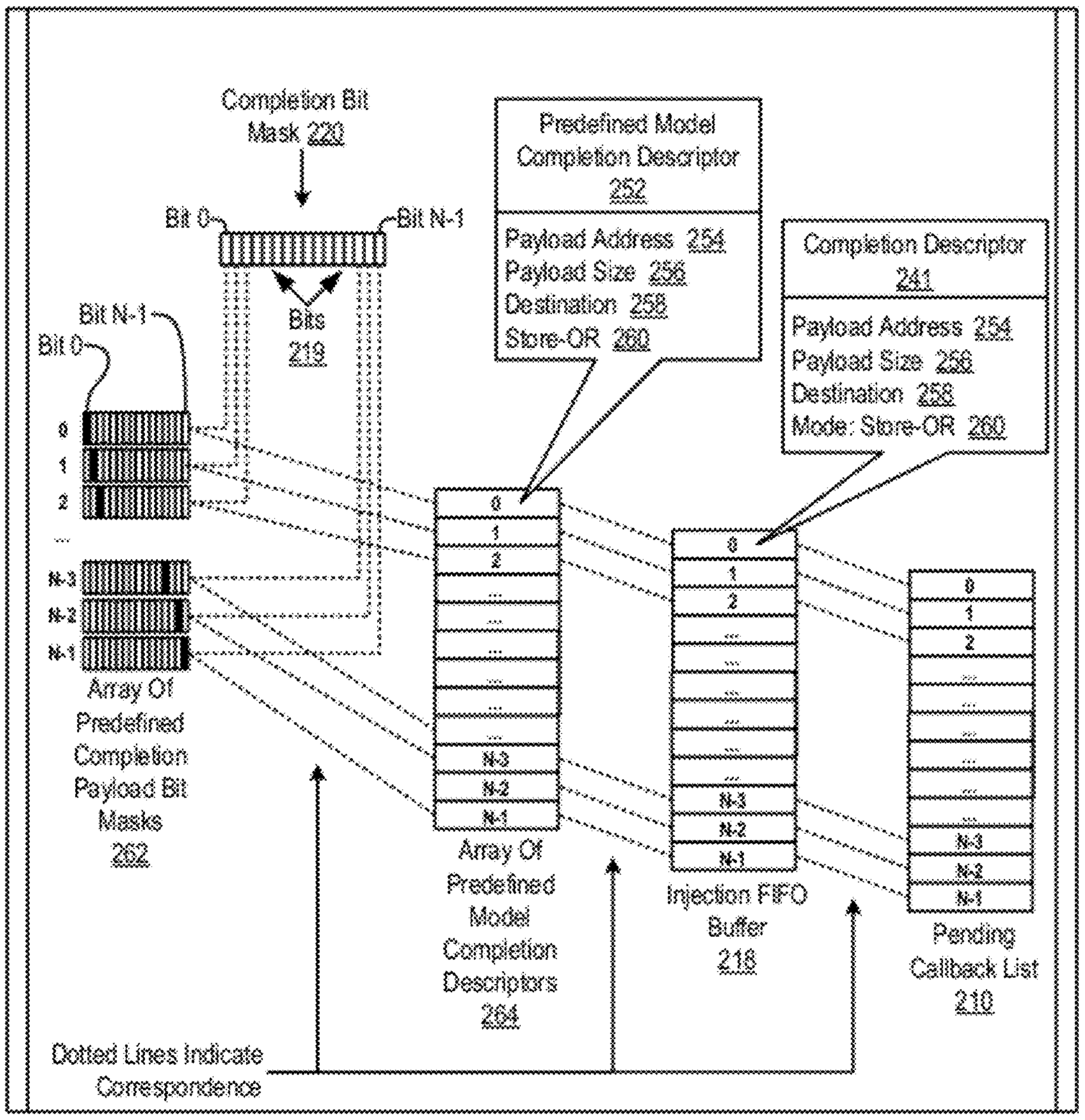

Completion Bit Mask 220
Bit 0
Bit N-1
Bits 219
Predefined Model Completion Descriptor 252
Payload Address 254
Payload Size 256
Destination 258
Store-OR 260
Completion Descriptor 241
Payload Address 254
Payload Size 256
Destination 258
Mode: Store-OR 260
Array Of Predefined Completion Payload Bit Masks 262
Array Of Predefined Model Completion Descriptors 264
Injection FIFO Buffer 218
Pending Callback List 210
Dotted Lines Indicate Correspondence

FIG. 4

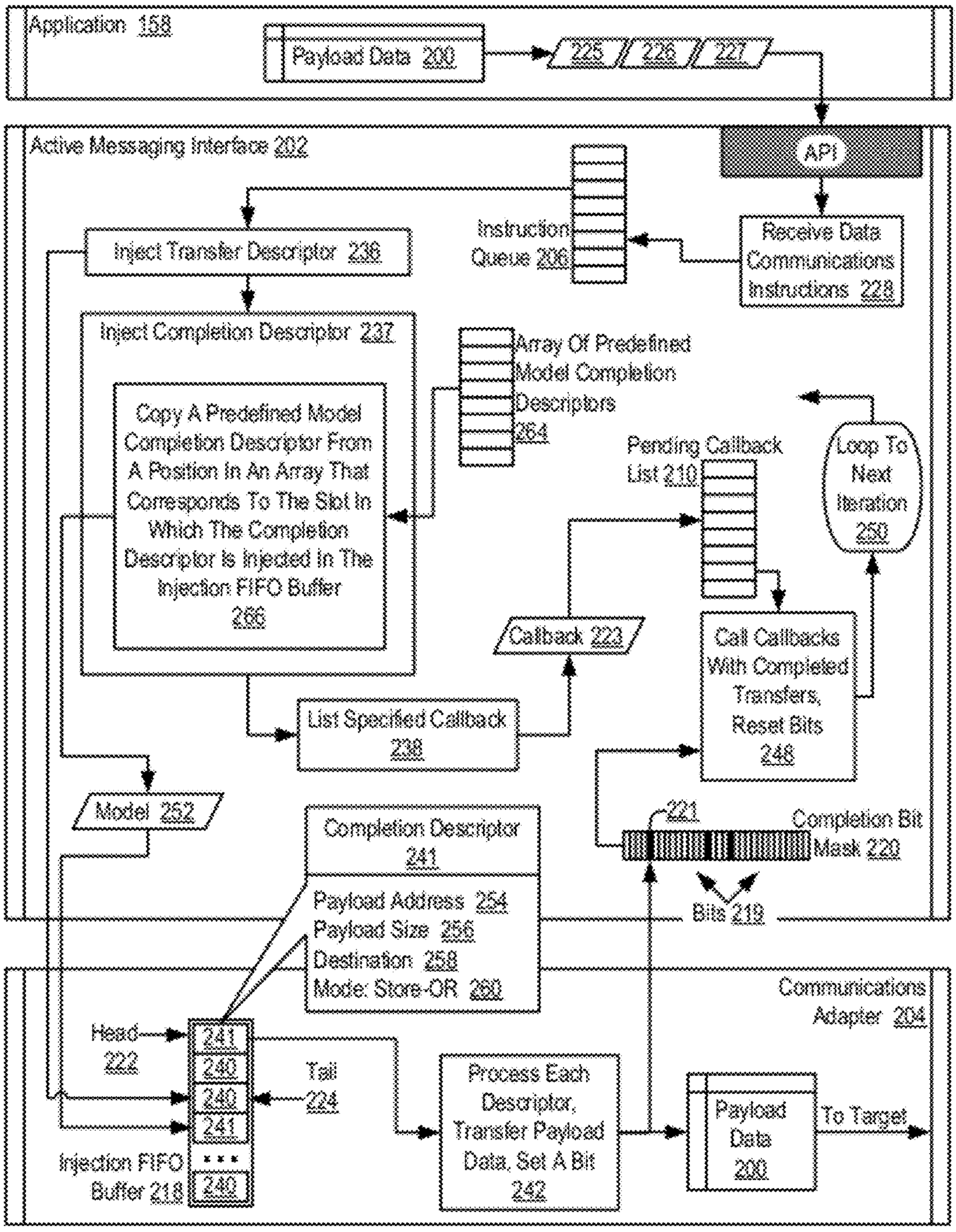

Application 158
Payload Data 200
225
226
227
Active Messaging Interface 202
API
Inject Transfer Descriptor 236
Instruction Queue 206
Receive Data Communications Instructions 228
Inject Completion Descriptor 237
Copy A Predefined Model Completion Descriptor From A Position In An Array That Corresponds To The Slot In Which The Completion Descriptor Is Injected In The Injection FIFO Buffer 266
Array Of Predefined Model Completion Descriptors 264
Pending Callback List 210
Loop To Next Iteration 250
Callback 223
Call Callbacks With Completed Transfers, Reset Bits 248
List Specified Callback 238
Model 252
Completion Descriptor 241
221
Completion Bit Mask 220
Payload Address 254
Payload Size 256
Destination 258
Mode: Store-OR 260
Bits 219
Communications Adapter 204
Head 222
241
240
240
241
Injection FIFO Buffer 218
240
Tail 224
Process Each Descriptor, Transfer Payload Data, Set A Bit 242
Payload Data 200
To Target

FIG. 5

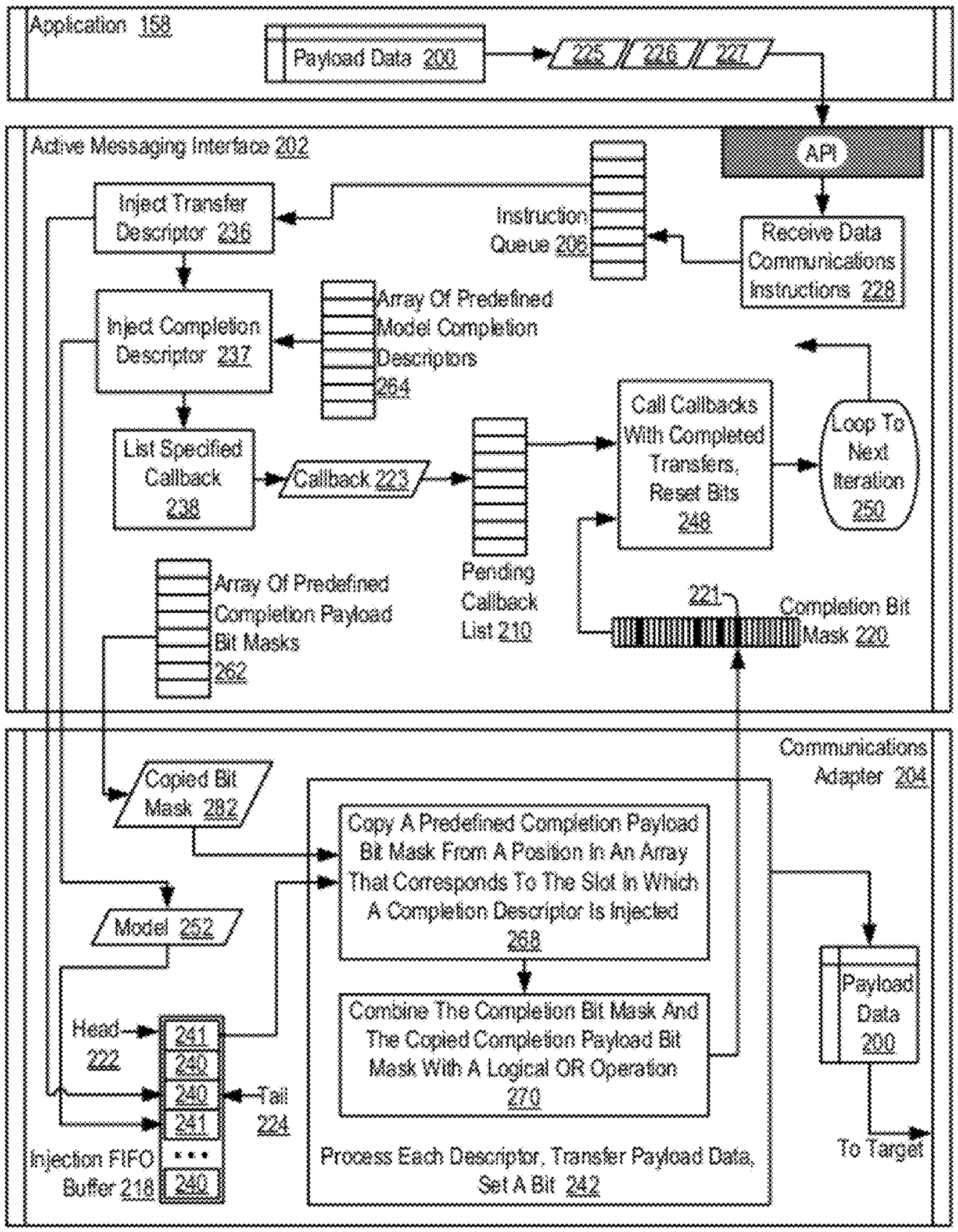

Application 158
Payload Data 200
225
226
227
Active Messaging Interface 202
API
Inject Transfer Descriptor 236
Instruction Queue 208
Receive Data Communications Instructions 228
Inject Completion Descriptor 237
Array Of Predefined Model Completion Descriptors 264
List Specified Callback 238
Callback 223
Call Callbacks With Completed Transfers, Reset Bits 248
Loop To Next Iteration 250
Array Of Predefined Completion Payload Bit Masks 262
Pending Callback List 210
221
Completion Bit Mask 220
Communications Adapter 204
Copied Bit Mask 282
Copy A Predefined Completion Payload Bit Mask From A Position in An Array That Corresponds To The Slot In Which A Completion Descriptor is Injected 268
Model 252
Payload Data 200
Head 222
241
240
240
241
Tail 224
Injection FIFO Buffer 218
240
Combine The Completion Bit Mask And The Copied Completion Payload Bit Mask With A Logical OR Operation 270
To Target
Process Each Descriptor, Transfer Payload Data, Set A Bit 242

FIG. 6

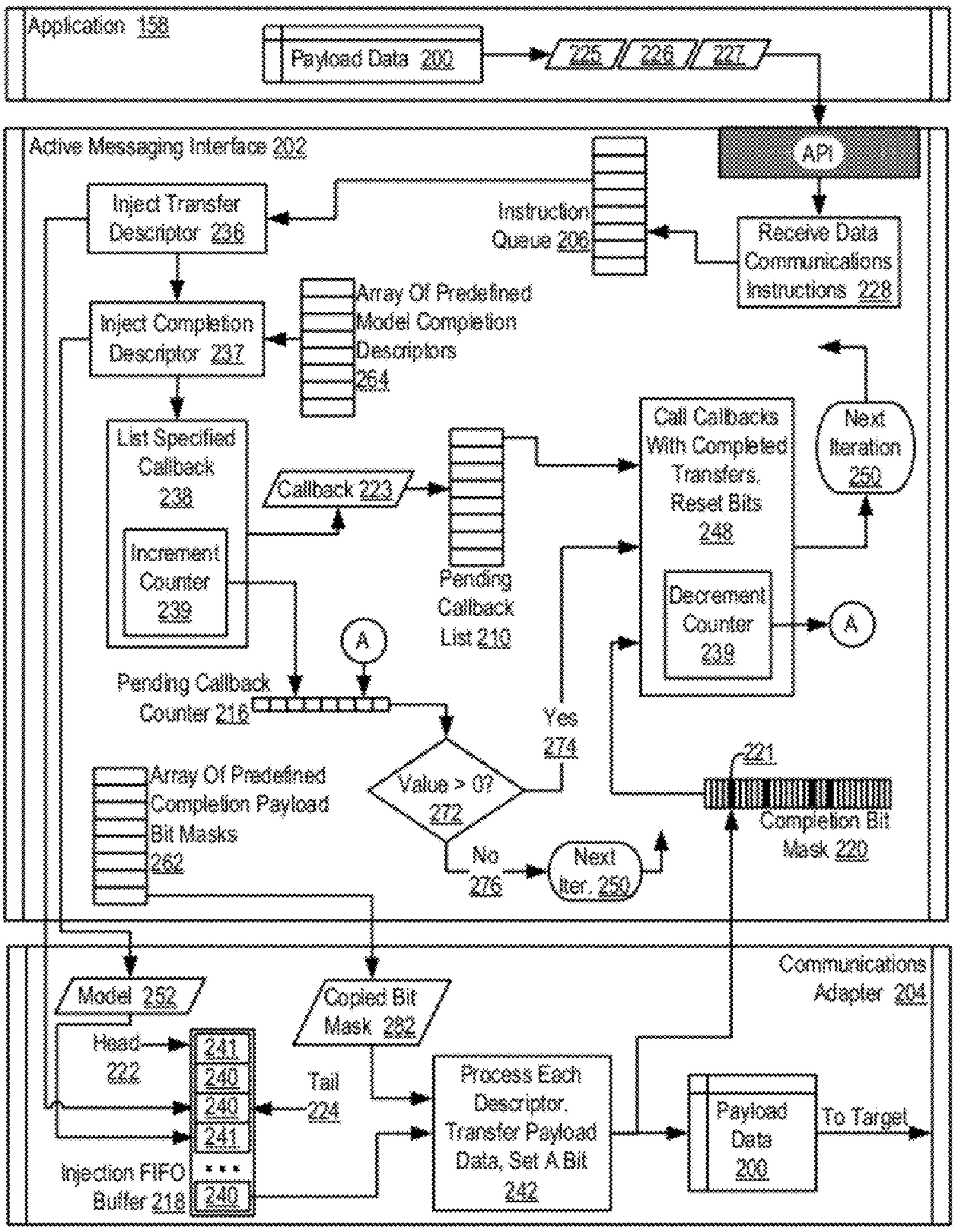

Application 158
Payload Data 200
225
226
227
Active Messaging Interface 202
API
Inject Transfer Descriptor 238
Instruction Queue 206
Receive Data Communications Instructions 228
Inject Completion Descriptor 237
Array Of Predefined Model Completion Descriptors 264
List Specified Callback 238
Increment Counter 239
Callback 223
Pending Callback List 210
Call Callbacks With Completed Transfers, Reset Bits 248
Next Iteration 250
Decrement Counter 239
A
Pending Callback Counter 216
Yes 274
Value > 0? 272
No 276
Next Iter. 250
221
Completion Bit Mask 220
Array Of Predefined Completion Payload Bit Masks 262
Model 252
Copied Bit Mask 282
Communications Adapter 204
Head 222
Tail 224
241
240
240
241
240
Injection FIFO Buffer 218
Process Each Descriptor, Transfer Payload Data, Set A Bit 242
Payload Data 200
To Target

FIG. 7

# COMPLETION PROCESSING FOR DATA COMMUNICATIONS INSTRUCTIONS

## STATEMENT REGARDING FEDERALLY SPONSORED RESEARCH OR DEVELOPMENT

**[0001]** This invention was made with Government support under Contract No. B554331 awarded by the Department of Energy. The Government has certain rights in this invention.

## BACKGROUND OF THE INVENTION

**[0002]** 1. Field of the Invention

**[0003]** The field of the invention is data processing, or, more specifically, methods, apparatus, and products for completion processing for data communications instructions in a distributed computing environment.

**[0004]** 2. Description of Related Art

**[0005]** The development of the EDVAC computer system of 1948 is often cited as the beginning of the computer era. Since that time, computer systems have evolved into extremely complicated devices. Today's computers are much more sophisticated than early systems such as the EDVAC. Computer systems typically include a combination of hardware and software components, application programs, operating systems, processors, buses, memory, input/output devices, and so on. As advances in semiconductor processing and computer architecture push the performance of the computer higher and higher, more sophisticated computer software has evolved to take advantage of the higher performance of the hardware, resulting in computer systems today that are much more powerful than just a few years ago.

**[0006]** Data communications is an area of computer technology that has experienced advances, and modes of data communications today effectively implement distributed computing environments. In the 1990s, a consortium that included Apollo Computer (later part of Hewlett-Packard), IBM, Digital Equipment Corporation, and others developed a software system that was named 'Distributed Computing Environment.' That software system is mentioned here for the sake of clarity to explain that the term 'distributed computing environment' as used in this specification does not refer that software product from the 1990s. As the term is used here, 'distributed computing environment' refers to any aggregation of computers or compute nodes coupled for data communications through a system-level messaging layer in their communications protocol stacks, where the system-level messaging layer provides 'active' messaging, messaging with callback functions. Implementations of such system-level messaging include messaging layers in client-server architectures, messaging layers in Symmetric Multi-Processing ('SMP') architectures with Non-Uniform Memory Access ('NUMA'), and messaging layers in parallel computers, including Beowulf clusters and even supercomputers with many compute node coupled for data communications through such system-level messaging. Common implementations of system-level messaging for parallel processing include the well known Message Passing Interface ('MPI') and the Parallel Virtual Machine ('PVM'). Both of these permit the programmer to divide a task among a group of networked computers, and collect the results of processing. Examples of MPI implementations include OpenMPI and MPICH. These and others represent examples of implementations of system-level messaging that can be improved for completion processing for data communications instructions in a distributed computing environment according to embodiments of the present invention.

**[0007]** Parallel computing is another area of computer technology that has experienced advances. Parallel computing is the simultaneous execution of the same application (split up and specially adapted) on multiple processors in order to obtain results faster. Parallel computing is based on the fact that the process of solving a problem often can be divided into smaller jobs, which may be carried out simultaneously with some coordination. Parallel computing expands the demands on middleware messaging beyond that of other architectures because parallel computing includes collective operations, operations that are defined only across multiple compute nodes in a parallel computer, operations that require, particularly in supercomputers, massive messaging at very high speeds. Examples of such collective operations include BROADCAST, SCATTER, GATHER, AND REDUCE operations.

**[0008]** Many data communications network architectures are used for message passing among nodes in parallel computers. Compute nodes may be organized in a network as a 'torus' or 'mesh,' for example. Also, compute nodes may be organized in a network as a tree. A torus network connects the nodes in a three-dimensional mesh with wrap around links. Every node is connected to its six neighbors through this torus network, and each node is addressed by its x,y,z coordinate in the mesh. In a tree network, the nodes typically are connected into a binary tree: each node has a parent and two children (although some nodes may only have zero children or one child, depending on the hardware configuration). In computers that use a torus and a tree network, the two networks typically are implemented independently of one another, with separate routing circuits, separate physical links, and separate message buffers.

**[0009]** A torus network lends itself to point to point operations, but a tree network typically is inefficient in point to point communication. A tree network, however, does provide high bandwidth and low latency for certain collective operations, message passing operations where all compute nodes participate simultaneously, such as, for example, an allgather.

**[0010]** There is at this time a general trend in computer processor development to move from multi-core to many-core processors: from dual-, tri-, quad-, hexa-, octo-core chips to ones with tens or even hundreds of cores. In addition, multi-core chips mixed with simultaneous multithreading, memory-on-chip, and special-purpose heterogeneous cores promise further performance and efficiency gains, especially in processing multimedia, recognition and networking applications. This trend is impacting the supercomputing world as well, where large transistor count chips are more efficiently used by replicating cores, rather than building chips that are very fast but very inefficient in terms of power utilization.

**[0011]** At the same time, the network link speed and number of links into and out of a compute node are dramatically increasing. IBM's BlueGene/Q™ supercomputer, for example, will have a five-dimensional torus network, which implements ten bidirectional data communications links per compute node—and BlueGene/Q will support many thousands of compute nodes. To keep these links filled with data, DMA engines are employed, but increasingly, the HPC community is interested in latency. In traditional supercomputers with pared-down operating systems, there is little or no multitasking within compute nodes. When a data communications link is unavailable, a task typically blocks or 'spins' on a data

transmission, in effect, idling a processor until a data transmission resource becomes available. In the trend for more powerful individual processors, such blocking or spinning has a bad effect on latency.

**[0012]** Of course if an application blocks or 'spins' on a data communications program, then the application is advised immediately when the transfer of data pursuant to the instruction is completed, because the application cease further processing until the instruction is completed. But that benefit comes at the cost of the block or the spin during a period of time when a high performance application really wants to be doing other things, not waiting on input/output. There is therefore a trend in the technology of large scale messaging toward attenuating this need to spin on a data communications resource waiting for completion of a data transfer. There is a trend toward supporting non-blocking data communications instructions that allow an application to fire-and-forget an instruction and check later with some infrastructure to confirm that the corresponding data transfer has actually been completed. The trend is to track data transfers with message sequence numbers stored temporarily in communications buffers in messaging infrastructure. If a message can be immediately completed, its sequence number can be flagged as completed, and the application can call down into the messaging infrastructure to figure out whether the message data has been sent. For messages that take more time, a completion descriptor can be created and marked later to advise the application when a transfer is completed. All these prior art methods of completion processing for data communications instructions, however, require significant data processing overheads, maintenance of additional data structures and data, additional system calls from the application to check on instruction completion.

## SUMMARY OF THE INVENTION

**[0013]** Methods, apparatus, and computer program products are described for completion processing of data communications instructions in a distributed computing environment, the distributed computing environment including a plurality of computers coupled for data communications through communications adapters and an active messaging interface ('AMI'), including injecting, by the AMI for each of a sequence of data communications instructions into a slot in an injection FIFO buffer of a data communication adapter, a transfer descriptor specifying to the communications adapter a transfer of payload data according to each data communications instruction, at least some of the instructions specifying callback functions; injecting by the AMI a completion descriptor for each instruction that specifies a callback function into the next slot after that instruction's transfer descriptor in the injection FIFO buffer, the slot in which the completion descriptor is injected having a corresponding slot in a pending callback list; listing, by the AMI in the corresponding slot in the pending callback list for each data communications instruction that specifies a callback function, the callback function specified by that instruction; processing by the communications adapter each descriptor in the injection FIFO buffer, including transferring payload data as specified by each transfer descriptor and setting, as payload data for each completion descriptor, a bit that corresponds in a completion bit mask to the slot in the FIFO where the completion descriptor was injected, the completion bit mask including a plurality of bits, each bit corresponding to a slot in the injection FIFO buffer; and calling by the AMI any callback functions in the pending callback list for which transfers of payload data have been completed as indicated by set bits in the completion bit mask.

**[0014]** The foregoing and other objects, features and advantages of the invention will be apparent from the following more particular descriptions of example embodiments of the invention as illustrated in the accompanying drawings wherein like reference numbers generally represent like parts of example embodiments of the invention.

## BRIEF DESCRIPTION OF THE DRAWINGS

**[0015]** FIG. **1** sets forth a functional block diagram of an example distributed computing environment that implements completion processing for data communications instructions according to embodiments of the present invention.

**[0016]** FIG. **2** sets forth a block diagram of an example protocol stack useful in apparatus that implements completion processing for data communications instructions in a distributed computing environment according to embodiments of the present invention.

**[0017]** FIGS. **3** and **5-7** set forth flow charts illustrating example methods of completion processing for data communications instructions in a distributed computing environment according to embodiments of the present invention.

**[0018]** FIG. **4** sets forth a block diagram of example structural elements useful in completion processing for data communications instructions in a distributed computing environment according to embodiments of the present invention.

## DETAILED DESCRIPTION OF EXAMPLE EMBODIMENTS

**[0019]** Example methods, apparatus, and products for completion processing of data communications instructions in a distributed computing environment according to embodiments of the present invention are described with reference to the accompanying drawings, beginning with FIG. **1**. FIG. **1** sets forth a functional block diagram of an example distributed computing environment (**122**) that implements completion processing for data communications instructions according to embodiments of the present invention. The distributed computing environment (**122**) of FIG. **1** includes several computers, an origin computer (**222**), a target computer (**224**), and other computers (**106**), all of which are coupled for data communications through communications adapters (**203**, **204**, **205**) and an active messaging interface ('AMI') (**202**). For ease of illustration, only the origin computer (**222**) and the target computer (**224**) are illustrated in detail with the communications adapters (**203**, **204**, **205**) and the AMI (**202**), but the other computers (**106**) also are so equipped.

**[0020]** The origin and target computers (**222**, **224**) in the example of FIG. **1** include one or more computer processors (**164**) or 'CPUs' as well as random access memory (**168**) ('RAM'). Each processor (**164**) can support multiple hardware compute cores (**165**), and each such core can in turn support multiple threads of execution, hardware threads of execution as well as software threads. Each processor (**164**) is connected to RAM (**168**) through a high-speed memory bus (**166**)—and through a high-speed front side bus (**162**), a bus adapter (**194**), and an expansion bus (**160**) to other components of the computer. Stored in RAM (**168**) is an application program (**158**), a module of computer program instructions that carries out user-level data processing using linear, SMP, or parallel algorithms that include data communications

among the computers in the distributed computing environment, including issuing data communications instructions to the AMI (**202**).

**[0021]** Also shown stored in RAM (**168**) is a the AMI (**202**) itself, a module of automated computing machinery that carries out completion processing for data communications instructions in a distributed computing environment according to embodiments of the present invention. An AMI (**202**) can be developed from scratch to carry out completion processing for data communications instructions in a distributed computing environment according to embodiments of the present invention, using a traditional programming language such as the C programming language or C++, for example, and using traditional programming methods to write communications routines that send and receive data among computers in the distributed computing environment through data communications networks or shared-memory transfers. Such an AMI developed from scratch can expose to applications an entirely new application programming interface ('API'). As an alternative to an AMI developed from scratch, an AMI (**202**) can expose a traditional API, such as MPI's API, to the application (**158**) so that the application can gain the benefits of an AMI with no need to recode the application. As an alternative to development from scratch, however, existing prior art system-level messaging modules may be improved to carry out completion processing for data communications instructions in a distributed computing environment according to embodiments of the present invention, existing modules that already implement a traditional interface. Examples of prior-art system-level messaging modules that can be improved to implement completion processing for data communications instructions in a distributed computing environment according to embodiments of the present invention include such parallel communications libraries as the traditional 'Message Passing Interface' ('MPI') library, the 'Parallel Virtual Machine' ('PVM') library, MPICH, and the like. In the example of FIG. **1**, the AMI (**202**) is represented in RAM (**168**). Readers will recognize, however, that the representation of the AMI in RAM is a convention for ease of explanation rather than a limitation of the present invention, because the AMI in fact can be implemented partly as software or firmware and hardware—or even, at least in some embodiments, entirely in hardware.

**[0022]** Also stored in RAM (**168**) is an operating system (**154**). An operating system is a computer software component that is responsible for execution of applications programs and for administration of access to computer resources, memory, processor time, and I/O functions, on behalf of application programs. Operating systems useful for completion processing for data communications instructions in a distributed computing environment according to embodiments of the present invention include UNIX™ Linux™ Microsoft XP™, AIX™ IBM's i5/OS™ and others as will occur to those of skill in the art. The application (**168**), the AMI (**202**), and the operating system (**154**) in the example of FIG. **1** are shown in RAM (**168**), but many components of such data processing modules typically are stored in non-volatile memory also, such as, for example, on a disk drive (**170**).

**[0023]** The origin computer (**222**) of FIG. **1** includes disk drive adapter (**172**) coupled through expansion bus (**160**) and bus adapter (**194**) to the processor (**164**) and other components of the computer (**222**). Disk drive adapter (**172**) connects non-volatile data storage to the computer (**222**) in the form of disk drive (**170**). Disk drive adapters useful in computers for completion processing for data communications instructions in a distributed computing environment according to embodiments of the present invention include Integrated Drive Electronics ('IDE') adapters, Small Computer System Interface ('SCSI') adapters, and others as will occur to those of skill in the art. Non-volatile computer memory also may be implemented as an optical disk drive, electrically erasable programmable read-only memory (so-called 'EEPROM' or 'Flash' memory), RAM drives, and so on, as will occur to those of skill in the art.

**[0024]** The example origin computer (**222**) of FIG. **1** includes one or more input/output ('I/O') adapters (**178**). I/O adapters implement user-oriented input/output through, for example, software drivers and computer hardware for controlling output to display devices such as computer display screens, as well as user input from user input devices (**181**) such as keyboards and mice. The example computer (**222**) of FIG. **1** includes a video adapter (**120**), which is an example of an I/O adapter specially designed for graphic output to a display device (**180**) such as a display screen or computer monitor. Video adapter (**120**) is connected to processors (**164**) through a high speed video bus (**164**), bus adapter (**194**), and the front side bus (**162**), which is also a high speed bus.

**[0025]** The example target and origin computers (**222**, **224**) of FIG. **1** include communications adapters (**203**, **204**, **205**) for data communications with other computers through a data communications network (**100**) or a segment of shared memory (**124**). Such data communications may be carried out serially through RS-232 connections, through external buses such as a Universal Serial Bus ('USB'), through data communications data communications networks such as Internet Protocol ('IP') data communications networks, and in other ways as will occur to those of skill in the art. Communications adapters implement the hardware level of data communications through which one computer sends data communications to another computer, directly, through shared memory, or through a data communications network. Examples of communications adapters useful for completion processing for data communications instructions in a distributed computing environment according to embodiments of the present invention include modems for wired dial-up communications, Ethernet (IEEE 802.3) adapters for wired data communications network communications, and 802.11 adapters for wireless data communications network communications. In the particular example of FIG. **1**, communications adapters (**203**, **204**, **205**) adapt computers for communications through a network (**100**). Examples of networks useful for data communications in a distributed computing environment according to embodiments of the present invention include Infiniband™, Gigabit Ethernet™, Fibre Channel™, PCI Express™, Serial ATA™, and others.

**[0026]** The communications adapters in the example of FIG. **1** include direct memory access ('DMA') adapters (**204**), modules of automated computing machinery that implement, through communications with other DMA adapters on other computers direct memory access to and from memory on its own computer as well as memory on other computers. Direct memory access is a way of reading and writing to and from memory among computers with reduced operational burden on computer processors (**164**); a CPU initiates a DMA transfer, but the CPU does not execute the DMA transfer. A DMA transfer essentially copies a block of memory from one computer to another, or between RAM segments of applications

on the same computer, from an origin to a target for a PUT operation, from a target to an origin for a GET operation, for example.

**[0027]** Also in the example of FIG. **1**, communications adapters (**205**) adapt computers for communications through a segment of shared memory (**124**). In the example of FIG. **1**, each processor or compute core has uniform access to the RAM (**168**) on the same computer, so that accessing a segment of shared memory is equally fast regardless where the shared segment is located in physical memory. In some embodiments, however, modules of physical memory are dedicated to particular processors, so that a processor may access local memory quickly and remote memory more slowly, a configuration referred to as a Non-Uniform Memory Access or 'NUMA.' In such embodiments, a segment of shared memory (**124**) can be configured locally for one endpoint and remotely for another endpoint—or remotely from both endpoints of a communication. In an embodiment, the origin computer (**222**) and the target computer (**224**) are both compute cores on the same compute node in a parallel computer, and, in that circumstance at least, a segment of shared memory (**124**) can be local to both the origin computer (**222**) and the target computer (**224**). From the perspective of an origin computer transmitting data through a segment of shared memory that is configured remotely with respect to the origin endpoint, transmitting data through the segment of shared memory will appear slower that if the segment of shared memory were configured locally with respect to the origin—or if the segment were local to both the origin and the target. The shared memory communications adapter (**205**) presents a similar interface to the AMI (**202**) as do the other adapters (**203**, **204**), including availability of an injection FIFO buffer (**218**). In embodiments where communications through a shared memory segment is available, however, it will often be faster than other methods.

**[0028]** The origin computer (**222**) and the target computer (**224**) are so labeled in this example because the origin computer is described as executing data communications instructions and therefore originating data transfers and the target computer is described as a subject of data communications instructions. The origin/target distinction does not describe the direction of data flow. A DMA PUT instruction transfers data from the origin computer to the target computer; a DMA GET instruction transfers data in the opposite direction from the target to the origin. In addition, the description here of only one target and one origin is not a limitation. In processing collective BROADCAST as a data communications instruction, a root process on an origin computer can transfer data to a large plurality of targets, including, for example, all of the computers (**222**, **224**, **106**) in the distributed computing environment—including treating itself as one of the targets. Similarly, in a collective GATHER, origin processes on all the computers in the distributed computing environment can transfer data to a single root process on one origin computer. In client/server, SMP, peer-to-peer, and other architectures, multiple origin computers send and receive message data among multiple target computers through an AMI.

**[0029]** The origin computer (**222**) in the example of FIG. **1** functions generally to carry out completion processing for data communications instructions in a distributed computing environment by receiving in the AMI (**202**) from an application (**158**) a sequence (**225** . . . **227**) of data communications instructions. In an embodiment, the application calls a function in an API that is exposed by the AMI to insert or post the instructions into an instruction queue in the AMI. In this way, the application's call to the API function is non-blocking. That is, the application is not required to block, spin, or otherwise wait for completion of the processing of the data communications instructions. The application inserts or posts an instruction into the queue, continues with other processing, and is informed of instruction completion by the AMI through a done callback.

**[0030]** Each instruction specifies a transfer of payload data (**200**) among computers in the distributed computing environment, and at least one of the instructions specifies a callback function. Examples of data communications instructions amendable to, or that can be improved to work with, completion processing according to embodiments of the present invention include the following, as well as others that will occur to those of skill in the art:

**[0031]** rendezvous network-based SEND instructions in which both origin and target endpoints communicate and participate in a data transfer, good for longer messages, typically composed of handshakes transferring header information followed by packet switched messaging or DMA operations to transfer payload data,

**[0032]** eager network-based SEND instructions in which only the origin or root computer conducts a data transfer, merely informing the target that the transfer has occurred, and requiring no communications or other participation from the target,

**[0033]** rendezvous SEND instructions with operations conducted, not through a network, but through shared memory, in which both the origin and target communicate and participate in a data transfer,

**[0034]** eager SEND instructions conducted, not through a network, but through shared memory, in which only the origin or root conducts a data transfer, merely informing targets that the transfer has occurred, but requiring no communications or other participation from the targets,

**[0035]** network-based DMA PUT instructions, useful for fast transfers of small messages, sometimes containing header data and payload data in a single transfer or packet—DMA algorithms also can be used as components of other instructions—as for example a SEND instruction that does an origin-target handshake and then conducts payload transfers with PUTs,

**[0036]** DMA PUT instructions with transfers through shared memory, again useful for fast transfers of small messages, sometimes containing header data and payload data in a single transfer or packet—DMA instructions also can be used as components of other algorithms—as for example a SEND instruction that does an origin-target handshake through a segment of shared memory and then conducts payload transfers with PUTs,

**[0037]** data communications instructions based on DMA GET operations, either networked or through shared memory, and

**[0038]** data communications instructions that include eager or rendezvous RECEIVE operations, either with send-side matching of SENDs or with receive-side matching.

**[0039]** The term 'payload' distinguishes header data and the like in data communications. The payload data (**200**) is specified typically with a buffer memory address and a quantity, for example, at memory address SendBuffer find one

kilobyte of payload data; the location and quantity of payload data as well as any callback functions are provided by the application (**158**) as parameters of the data communications instructions (**225** . . . **227**). A 'callback function' is often referred to in this specification simply as a 'callback.' Callback functions include dispatch callbacks as well as done callbacks. A dispatch callback is a function to be called upon receipt of a data communications instruction. A done callback is a function to be called upon completion of the transfer of payload data as specified by a data communications instruction. Except as otherwise stated in context, discussion and description of a callback in this specification is a description of a done callback, so that the term 'callback' and 'done callback' are generally synonyms, unless otherwise stated.

**[0040]** The origin computer (**222**) in the example of FIG. **1** through its AMI (**202**) also injects, for each data communications instruction (**225** . . . **227**) into a slot in an injection FIFO buffer (**218**) of a data communication adapter (**203**, **204**, **205**), a transfer descriptor (**240**). The transfer descriptor specifies to the communications adapter the transfer of payload data, and the slot in the injection FIFO buffer (**218**) has a corresponding slot in a pending callback list (**210**). The term 'injection' connotes the 'injection' of transfer data into a data communications resource, a network, a shared memory, and the like, for actual transport to a target. A transfer descriptor provides a description of a data communications instruction that is recognizable or administrable by lower level data communications resources, DMA adapters, other communications adapters, and the like. The origin computer (**222**) also injects by the AMI (**202**) a completion descriptor (**241**) for each instruction that specifies a callback function into the next slot after that instruction's transfer descriptor in the injection FIFO buffer (**218**), the slot in which the completion descriptor is injected having a corresponding slot in a pending callback list (**210**). The origin computer (**222**) also lists, through the AMI (**202**) in the corresponding slot in the pending callback list (**210**) for each data communications instruction, any callback function specified by that instruction.

**[0041]** The communications adapter (here, one of **203**, **204**, **205**) that received the descriptors (**240**, **241**) processes each descriptor in its injection FIFO buffer (**218**), including transferring payload data as specified by each transfer descriptor and setting, as payload data for each completion descriptor, a bit that corresponds in a completion bit mask (**220**) to the slot in the FIFO where the completion descriptor was injected. The completion bit mask (**220**) is composed of bits, with each bit corresponding to a slot in the injection FIFO buffer (**218**). The origin computer (**222**) through its AMI (**202**) also calls any callback functions in the pending callback list (**210**) for which transfers of payload data have been completed as indicated by set bits in the completion bit mask (**220**).

**[0042]** The arrangement of computers, communications adapters, and other devices making up the example distributed computing environment illustrated in FIG. **1** are for explanation, not for limitation. Data processing systems useful for completion processing for data communications instructions in a distributed computing environment according to various embodiments of the present invention may include additional servers, routers, other devices, and peer-to-peer architectures, not shown in FIG. **1**, as will occur to those of skill in the art. Networks in such data processing systems may support many data communications protocols, including for example TCP (Transmission Control Protocol), IP (Internet Protocol), HTTP (HyperText Transfer Protocol), WAP (Wireless Access Protocol), HDTP (Handheld Device Transport Protocol), and others as will occur to those of skill in the art. Various embodiments of the present invention may be implemented on a variety of hardware platforms in addition to those illustrated in FIG. **1**.

**[0043]** For further explanation, FIG. **2** sets forth a block diagram of an example protocol stack useful in apparatus that implements completion processing for data communications instructions in a distributed computing environment according to embodiments of the present invention. The example protocol stack of FIG. **2** includes a hardware layer (**214**), a system messaging layer (**212**), and an application layer (**208**). For ease of explanation, the protocol layers in the example stack of FIG. **2** are shown connecting an origin computer (**222**) and a target computer (**224**), although it is worthwhile to point out that in embodiments, the origin computer and the target computer can be the same computer, because any particular transfer can be from an origin application on a computer to a target application on the same computer. This pattern would be very common, for example, in a supercomputer whose compute nodes operate multi-threaded. Every thread of execution on such a computer can function as both an origin or a target for data transfers through an AMI, and both the origin and its target can be located on the same computer. So an origin computer (**222**) and its target computer (**224**) can in fact, and often will, be the same computer.

**[0044]** The application layer (**208**) provides communications among applications (**158**) running on the computers (**222**, **224**) by invoking functions in an Active Messaging Interface ('AMI') (**202**) installed on each computer. Applications may communicate messages by invoking functions of an application programming interface ('API') exposed by the AMI (**202**). The AMI can expose a novel, custom API, or the AMI can expose a traditional API, such as, for example, an API of an MPI library, to applications (**158**) so that the application can gain the benefits of an AMI, reduced network traffic, callback functions, and so on, with little or no need to recode the application.

**[0045]** The example protocol stack of FIG. **2** includes a system messaging layer (**212**) implemented here as an Active Messaging Interface or 'AMI' (**202**). The AMI provides system-level data communications functions that support messaging in the application layer (**208**) and the system messaging layer (**212**). Such system-level functions are typically invoked through an API exposed to the application (**158**) in the application layer (**208**).

**[0046]** The protocol stack of FIG. **2** includes a hardware layer (**214**) that defines the physical implementation and the electrical implementation of aspects of the hardware on the computers such as the bus, network cabling, connector types, physical data rates, data transmission encoding and many other factors for communications between the computers (**222**, **224**) on the physical network medium. In computers that implement completion processing for data communications instructions in a distributed computing environment according to embodiments of the present invention, the hardware layer includes communications adapters, including DMA adapters, and network links, including routers, packet switches, and the like. In this particular example, the hardware layer (**214**) in each computer includes a communication adapter (**204**).

**[0047]** The origin computer (**222**) in the example of FIG. **2** functions generally to carry out completion processing for data communications instructions in a distributed computing

environment by receiving in the AMI (**202**) from an application (**158**) a sequence (**225** . . . **227**) of data communications instructions. In an embodiment, the application (**158**) calls a function in an API that is exposed by the AMI to insert or post the instructions into an instruction queue (**206**) in the AMI. In this way, the application's call to the API function is non-blocking. The application (**158**) does not block or spin on the post to the instruction queue. Instead, the application inserts or posts an instruction into the queue (**206**), continues with other processing, and is eventually informed of instruction completion by the AMI through a done callback. Each instruction (**225** . . . **227**) specifies a transfer of payload data (**200**) among computers in a distributed computing environment, and some of the data communications instructions specify callback functions (**223**).

**[0048]** The origin computer (**222**) in the example of FIG. **2** also injects, by the AMI (**202**) for each data communications instruction (**225** . . . **227**) into a slot in an injection FIFO buffer (**218**) of a data communication adapter (**204**), a transfer descriptor (**240**). The origin computer (**222**) also injects by the AMI (**202**) a completion descriptor (**241**) for each instruction that specifies a callback function into the next slot after that instruction's transfer descriptor in the injection FIFO buffer (**218**), the slot in which the completion descriptor is injected having a corresponding slot in a pending callback list (**210**). The origin computer (**222**) also lists, through the AMI (**202**) in the corresponding slot in the pending callback list (**210**) for each data communications instruction, any callback function specified by that instruction.

**[0049]** 'FIFO' is an abbreviation of 'first-in-first-out' and connotes the fact that the communications adapter (**204**) processes its descriptors in the order in which they are placed in the injection FIFO buffer (**218**). The completion descriptor (**241**) as well as the transfer descriptor (**240**) both specify to the communications adapter (**204**) a transfer of payload data, although in the case of the completion descriptor (**241**), the payload is a bit mask that is delivered to the completion bit mask (**220**) in the AMI (**202**). The slot in the injection FIFO buffer (**218**) where the completion descriptor (**241**) is injected has a corresponding slot in a pending callback list (**210**); actually every slot in the injection FIFO buffer has a corresponding slot in the pending callback list. Both the injection FIFO buffer (**218**) and the pending callback list (**210**) are apportioned into N slots, here labeled 0 . . . N−1. The slots 'correspond' in that:

**[0050]** a pending callback for a completion descriptor in slot **0** of the injection FIFO buffer (**218**) is listed in slot **0** of the pending callback list (**210**),

**[0051]** a pending callback for a completion descriptor in slot **1** of the injection FIFO buffer (**218**) is listed in slot **1** of the pending callback list (**210**),

**[0052]** . . .

**[0053]** a pending callback for a completion descriptor in slot N−2 of the injection FIFO buffer (**218**) is listed in slot N−2 of the pending callback list (**210**), and

**[0054]** a pending callback for a completion descriptor in slot N−1 of the injection FIFO buffer (**218**) is listed in slot N−1 of the pending callback list (**210**).

**[0055]** The term 'pending' as used here indicates that a callback has been listed but its corresponding data transfer has not yet been completed. Each done callback is called only after completion of its corresponding data transfer, the transfer represented by a transfer descriptor in a slot in the injection FIFO buffer. The communications adapter (**204**) processes each descriptor in its injection FIFO buffer (**218**), including transferring payload data as specified by each transfer descriptor and setting, as payload data for each completion descriptor, a bit that corresponds in a completion bit mask (**220**) to the slot in the FIFO where the completion descriptor was injected. The completion bit mask (**2120**) is composed of bits, with each bit corresponding to a slot in the injection FIFO buffer (**218**). The origin computer (**222**) through its AMI (**202**) also calls any callback functions in the pending callback list (**210**) for which transfers of payload data have been completed as indicated by set bits in the completion bit mask (**220**).

**[0056]** For further explanation, FIG. **3** sets forth a flow chart illustrating an example method of completion processing for data communications instructions in a distributed computing environment according to embodiments of the present invention. FIG. **4** sets forth a block diagram of example structural elements useful in the method of FIG. **3**; the method of FIG. **3** therefore is described here with reference both to FIGS. **3** and **4**, using reference numbers from both FIGS. **3** and **4**. The method of FIG. **3** is implemented in a distributed computing environment like that described above with reference to FIG. **1**, a distributed computing environment that includes a plurality of computers coupled for data communications through communications adapters and through an active messaging interface ('AMI'). The AMI (**202**) is a module of automated computing machinery that operates iteratively to process data communications instructions (**225** . . . **227**) disposed in a queue (**206**) of data communications instructions.

**[0057]** The method of FIG. **3** includes receiving (**228**), in the AMI from an application (**158**) executing on a computer in a distributed computing environment, a sequence of data communications instructions (**225** . . . **227**). Each such instruction specifies a transfer of payload data (**200**) among computers in the distributed computing environment, and at least some of the instructions specify callback functions (**223**). In the example of FIG. **3**, the application (**158**) calls a function in an API that is exposed by the AMI to insert or post the instructions into an instruction queue (**206**) in the AMI. The application's call to the API function is non-blocking. That is, the application is not required to wait around for completion of the processing of the data communications instructions. The application inserts or posts an instruction into the queue, continues with other processing, and is informed of instruction completion by the AMI through a done callback. The function of receiving (**228**) data communication instructions is asynchronous with respect to the iterative processing carried out by the AMI, because the storage functionality of the instruction queue (**206**) disconnects the function of receiving (**228**) instructions from the iterative processing of the instructions within the AMI.

**[0058]** The method of FIG. **3** also includes determining (**230**) by the AMI (**202**) for each data communications instruction (**225** . . . **227**) that space is available in an injection FIFO buffer (**218**) of the communications adapter (**204**) for that instruction's transfer descriptor (**240**) and, if the instruction specifies a callback, space available for a completion descriptor (**241**). Determining (**230**) that space is available in the injection FIFO buffer (**218**), in the example of FIG. **3**, is carried out by first determining (**232**) that a slot is open in the injection FIFO buffer, or two slots if the pertinent instruction specifies a callback. The communication adapter (**204**) maintains buffer pointers: a head pointer (**222**) that points to the

'head' of the injection FIFO buffer, that is, the slot containing the descriptor currently being processed by the communication adapter, and a tail pointer (**224**) that points to the next slot available for injection of a descriptor. Comparing the values of these two pointers (**222**, **224**) yields the number of available slots in the injection FIFO buffer. If the value of the tail point is equal to the value of the head point, the tail has caught up to the head or the head has caught the tail, the number of available slots is 0, and processing in the AMI must pause until the communications adapter completes processing of the descriptor in the slot currently indicated by the head pointer, incrementing the head pointer, making that slot available for use. The communications adapter can maintain the number of available slots, so that the AMI can query the adapter for that information, or the AMI can examine the pointers (**222**, **224**) directly and make the comparison itself. If the pointers (**222**, **224**) indicate that no FIFO slot is open for a descriptor, the AMI continues processing (**233**) without injecting (**236**, **237**) a descriptor and without listing (**238**) any callback, instead proceeding directly to calling (**248**) any listed pending callbacks with completed transfers, resetting corresponding bits in the completion bit mask (**220**), and looping (**250**) around to the next iteration of processing in the AMI to determining (**230**) space available, inject (**236**, **237**) descriptors, listing (**238**) specified callbacks, and so on.

**[0059]** The fact that one or more slots are open in the injection FIFO buffer, however, does not necessarily mean that each such slot is available for a transfer descriptor. Each completion descriptor in the injection FIFO buffer has a callback in a corresponding slot in the pending callback list that possibly has not yet been called. In the course of processing operations, the communication adapter completes a data transfer according to a descriptor, increments the head pointer to point to the next slot, sets a bit in the completion bit mask if the descriptor was a completion descriptor, and moves on to work on the next descriptor, all without knowing whether any corresponding callback has been called. For each completion descriptor (**241**), if the corresponding callback has not been called, then it is premature for the AMI yet to use the recently vacated slot in the injection FIFO buffer. In the example of FIG. **3**, therefore, determining (**230**) that space is available in the injection FIFO buffer (**218**) also includes determining (**234**) that a corresponding slot contains a null value in the pending callback list. If the available slot in the FIFO buffer contained a transfer descriptor, its corresponding slot in the pending callback list will always be nulled. If the available slot in the FIFO buffer contained a completion descriptor, its corresponding slot in the pending callback list will be null if the callback has already been called, otherwise non-null, containing the callback function in the form of a pointer or index value. The corresponding slot in the pending callback list is a slot in the pending callback list that corresponds to a slot in the injection FIFO buffer to which the tail pointer (**224**) currently points. If the tail pointer (**224**) currently points to slot **0** in the injection FIFO buffer, then the corresponding slot to check is slot **0** in the pending callback list. If the tail pointer (**224**) currently points to slot **1** in the injection FIFO buffer, then the corresponding slot to check is slot **1** in the pending callback list. And so on. A null value in the corresponding slot of the pending callback list indicates that the corresponding slot in the injection FIFO buffer is actually available for injection, because either the corresponding callback has already been called—or the corresponding data communications instruction specified no callback so no callback was ever listed in that slot. Either way, if the pointers (**222**, **224**) indicate an open slot in the injection FIFO buffer and the corresponding slot in the pending callback list is null, then the open slot in the injection FIFO buffer is actually available for an injection of a descriptor from the AMI, either a transfer descriptor or a completion descriptor. If the corresponding slot in the pending callback list is non-null, it contains a listed callback that has not yet been called, and the AMI continues processing without injecting (**236**, **237**) a descriptor and without listing (**238**) any callback, instead proceeding directly to calling (**248**) any listed pending callbacks with completed transfers, looping (**250**) around to the next iteration to determine (**230**) whether space is available in the injection FIFO buffer (**218**), injecting (**236**, **237**) descriptors, listing (**238**) specified callbacks, and so on.

**[0060]** The method of FIG. **3** also includes injecting (**236**), by the AMI for each data communications instruction (**225** . . . **227**) into a slot in an injection FIFO buffer (**218**) of a data communication adapter (**204**), a transfer descriptor (**240**)—as well as injecting (**237**) by the AMI (**202**) a completion descriptor (**241**) for each instruction that specifies a callback function. The AMI checks (**280**) whether an instruction specifies a callback, and, if the instruction does not specify a callback (**281**), the AMI continues processing by calling (**248**) callbacks with completed data transfer, skipping for that instruction in this iteration the steps of injecting (**237**) a completion descriptor and listing (**238**) a specified callback. If the instruction does specify a callback (**280**, **279**), the AMI injects (**237**) a completion descriptor, lists (**238**) the specified callback, and continues with calling (**248**) callbacks with completed data transfers. The completion descriptor (**241**) is injected into the next slot after that instruction's transfer descriptor (**240**) in the injection FIFO buffer (**218**). The slot in which the completion descriptor (**241**) is injected has a corresponding slot in a pending callback list (**210**). Each slot in the injection FIFO buffer (**218**) has a corresponding slot in the pending callback list (**210**) where a corresponding callback function can be listed. In addition, in injecting (**236**, **237**) a descriptor, the AMI also increments the tail pointer (**224**) of the injection FIFO buffer (**218**) to point to a next slot in the injection FIFO buffer. This description is to a 'next slot' only, not to a 'next open slot.' The AMI need not test at this time whether the next slot is actually open; the AMI just increments the pointer (**224**) to the next slot. If it turn out later that that next slot is not open, that fact will be determined (**230**, **232**, **234**) on a next iteration through the functions of the AMI. For instructions that specify a callback, the tail pointer (**224**) is moved two slots, one for a transfer descriptor and another for the corresponding completion descriptor, but this processing step does not necessarily require two separate operations. The tail pointer can be incremented to point two slots further in a single operation after injection of a completion descriptor and its accompanying transfer descriptor.

**[0061]** The method of FIG. **3** also includes listing (**238**), by the AMI (**202**) in the corresponding slot in the pending callback list (**210**) for each data communications instruction, any callback function (**223**) specified by that instruction. In this context, the corresponding slot is the slot in the pending callback list that corresponds to the slot in the injection FIFO buffer where the instruction's completion descriptor is injected. Although some will not, many of the data communications instructions specify callbacks. When the AMI calls a callback, the AMI nulls the slot in the pending callback list where that callback was listed and resets the bit in the comple-

tion bit mask corresponding to that now null slot in the pending callback list. For a data communications instruction that does not specify a callback function, the slot in the pending callback list corresponding to the slot in the injection FIFO buffer where a transfer descriptor is injected for that instruction is left null.

**[0062]** The method of FIG. **3** also includes processing (**242**) by the communications adapter (**204**) each descriptor (**240**, **241**) in the injection FIFO buffer (**218**). In such processing, the communications adapter transfers payload data as specified by each transfer descriptor (**240**) in the injection FIFO buffer (**218**). The communications adapter (**204**) also sets, as payload data for each completion descriptor (**241**), a bit (**221**) that corresponds in a completion bit mask (**220**) to the slot in the FIFO where the completion descriptor was injected. The completion bit mask (**220**) is composed of a number of bits, with each bit corresponding to a slot in the injection FIFO buffer. In this example, the completion bit mask (**220**) is composed of N bits labeled 0 . . . N−1, and the injection FIFO buffer (**218**) is composed of N slots labeled 0 . . . N−1. Bit **0** in the completion bit mask (**218**) corresponds to slot **0** in the injection FIFO buffer (**218**), and setting bit **0** indicates that the communications adapter (**204**) has completed the transfer of payload data specified by the transfer descriptor (**240**) in slot N−1 of the injection FIFO buffer (**218**) and the presence of an as yet uncalled callback listed in slot **0** of the pending callback list (**210**). Bit **1** in the completion bit mask (**218**) corresponds to slot **1** in the injection FIFO buffer (**218**), and setting bit **1** indicates that the communications adapter (**204**) has completed the transfer of payload data specified by the transfer descriptor (**240**) in slot **0** of the injection FIFO buffer (**218**) and the presence of an as yet uncalled callback listed in slot **1** of the pending callback list (**210**). Bit **2** in the completion bit mask (**218**) corresponds to slot **2** in the injection FIFO buffer (**218**), and setting bit **2** indicates that the communications adapter (**204**) has completed the transfer of payload data specified by the transfer descriptor (**240**) in slot **1** of the injection FIFO buffer (**218**) and the presence of an as yet uncalled callback listed in slot **2** of the pending callback list (**210**). Bit N−1 in the completion bit mask (**218**) corresponds to slot N−1 in the injection FIFO buffer (**218**), and setting bit N−1 indicates that the communications adapter (**204**) has completed the transfer of payload data specified by the transfer descriptor (**240**) in slot N−2 of the injection FIFO buffer (**218**) and the presence of an as yet uncalled callback listed in slot N−1 of the pending callback list (**210**). And so on.

**[0063]** Also in embodiments, processing (**242**) the descriptors (**240**,**241**) in the injection FIFO buffer (**218**) includes incrementing, upon completion of processing for each descriptor, the head pointer (**222**) of the injection FIFO buffer (**218**) to point to a next descriptor (**240**, **241**) to be processed by the communications adapter (**204**). This element of processing is the same for transfer descriptors (**240**) and completion descriptors (**241**), and the head pointer is moved one slot for both. This motion of the head pointer contrasts somewhat with the motion of the tail pointer (**224**) in injecting descriptors for data communications instructions that specify callbacks, when the tail pointer is moved two slots, one for a transfer descriptor and one for its accompanying completion descriptor. For an instruction that does not specify a callback, the tail pointer moves only one slot, for the injection of only a transfer descriptor.

**[0064]** The method of FIG. **3** also includes calling (**248**) by the AMI any callback functions in the pending callback list (**210**) for which transfers of payload data have been completed. The set bits in the completion bit mask indicate which slots in the pending callback list contain callbacks for which transfers of payload data have been completed. In the drawings, the bit positions in bit masks that are colored black indicate set bits, and white indicates reset bits. 'Set' means set to logical TRUE, a binary value of '1'. 'Reset' means reset to logical FALSE, a binary value of '0.' The AMI scans the bits in the completion bit mask (**220**). If bit **0** in the completion bit mask (**220**) is set, the AMI calls the callback function listed in slot **0** of the pending callback list (**210**), and resets bit **0**. If bit **1** in the completion bit mask (**220**) is set, the AMI calls the callback function listed in slot **1** of the pending callback list (**210**), and resets bit **1**. If bit **2** in the completion bit mask (**220**) is set, the AMI calls the callback function listed in slot **2** of the pending callback list (**210**), and resets bit **2**. And so on.

**[0065]** For further explanation, FIG. **5** sets forth a flow chart illustrating a further example method of completion processing for data communications instructions in a distributed computing environment according to embodiments of the present invention. FIG. **4** sets forth a block diagram of example structural elements useful in the example method of FIG. **5**; the method of FIG. **5** therefore is described here with reference both to FIGS. **4** and **5**, using reference numbers from both FIGS. **4** and **5**.

**[0066]** The method of FIG. **5** is similar to the method of FIG. **3** in that it also is implemented in a distributed computing environment like that described above with reference to FIG. **1**, a distributed computing environment that includes a plurality of computers coupled for data communications through communications adapters and through an active messaging interface ('AMI'). The AMI (**202**) is a module of automated computing machinery that operates iteratively to process data communications instructions (**225** . . . **227**) received (**228**) from an application (**158**) and disposed in a queue (**206**) of data communications instructions. The method of FIG. **5** is also like the method of FIG. **3** in that it includes injecting (**236**) transfer descriptors (**240**); injecting (**237**) completion descriptors (**241**) for instructions that specify a callback (**223**); listing (**238**) in a pending callback list (**210**) any callback function (**223**) specified by an instruction; processing (**242**) descriptors (**240**, **241**) through the communications adapter (**204**), setting a bit (**221**) in a completion bit mask (**220**) for each completion descriptor processed; and calling (**248**) any callback functions in the pending callback list (**210**) for which transfers of payload data have been completed as indicated by set bits in the completion bit mask (**220**).

**[0067]** In the method of FIG. **5**, however, injecting (**237**) a completion descriptor (**241**) includes copying (**237**) a predefined model completion descriptor (**252**) from a position in an array (**264**) of predefined model completion descriptors that corresponds to the slot in which the completion descriptor (**241**) is injected in the injection FIFO buffer (**218**). The array (**264**) of predefined model completion descriptors includes predefined model completion descriptors (**252**) for all slots in the injection FIFO buffer. Each element of the array (**264**) of predefined model completion descriptors contains a predefined model completion descriptor (**252**) that is constructed to correspond with a slot in the injection FIFO buffer (**218**). The array (**264**) of predefined model completion descriptors contains N elements labeled 0 . . . N−1, and the injection FIFO buffer (**218**) is apportioned into N slots labeled 0 . . . N−1. Each slot in the injection FIFO buffer

corresponds with an element of the array (**264**) of predefined model completion descriptors, slot **0** corresponds with element **0**, slot **1** with element **1**, slot **2** with element **2**, and so on through slot N−1 and element N−1. Such correspondence means that the process of injecting a transfer descriptor can be reduced to a single memcopy( ) operation, an extremely fast mode of data processing. Injecting a completion descriptor into the $n^{th}$ slot in the injection FIFO buffer is simplified to a single memcopy( ) from the $n^{th}$ element of the array (**264**) of predefined model completion descriptors, thereby injecting a predefined model completion descriptor (**252**) to be used as the completion descriptor in the $n^{th}$ slot of the injection FIFO buffer.

**[0068]** Each predefined model completion descriptor (**252**) is predefined with member data elements describing a payload data address (**254**), payload data size (**256**), a target destination (**258**), and a transfer mode (**260**) that is specified in these examples as a store-OR operation, a store to a destination address combined with a logical OR operation. The payload data address (**254**) is predefined in each predefined model completion descriptor (**252**) to point to a corresponding element in an array (**262**) of predefined completion payload bit masks. Each element in the array (**262**) of predefined completion payload bit masks contains a predefined completion payload bit mask configured to correspond to a slot in the injection FIFO buffer and therefore also correspond to an element in the array (**264**) of predefined model completion descriptors and a slot in the pending callback list (**210**), correspondences indicated graphically by the dotted lines on FIG. **4**. The destination element (**258**) in each predefined model completion descriptor (**252**) is preconfigured to point to the beginning address of the completion bit mask (**220**), and the payload size (**256**) is preconfigured to the size of the completion bit mask (**220**), which is also the size of each element in the array (**262**) of predefined completion payload bit masks, so that each element in the array (**262**) of predefined completion payload bit masks will exactly fit the completion bit mask (**220**) for bitwise operations, a logical OR, for example, or a store-OR operation.

**[0069]** For further explanation, FIG. **6** sets forth a flow chart illustrating a further example method of completion processing for data communications instructions in a distributed computing environment according to embodiments of the present invention. FIG. **4** sets forth a block diagram of example structural elements useful in the example method of FIG. **6**; the method of FIG. **6** therefore is described here with reference both to FIGS. **4** and **6**, using reference numbers from both FIGS. **4** and **6**. The method of FIG. **6** is similar to the method of FIG. **5** in that it also is implemented in a distributed computing environment like that described above with reference to FIG. **1**, a distributed computing environment that includes a plurality of computers coupled for data communications through communications adapters and through an AMI. The method of FIG. **6** is also like the method of FIG. **5** in that it includes injecting (**236**) transfer descriptors (**240**); injecting (**237**) completion descriptors (**241**) for instructions that specify a callback (**223**); listing (**238**) in a pending callback list (**210**) any callback function (**223**) specified by an instruction; processing (**242**) descriptors (**240**, **241**) through the communications adapter (**204**), setting a bit (**221**) in a completion bit mask (**220**) for each completion descriptor processed; and calling (**248**) any callback functions in the pending callback list (**210**) for which transfers of payload data have been completed as indicated by set bits in the completion bit mask (**220**). Also in the method of FIG. **6**, like the method of FIG. **5**, the AMI injects (**237**) completion descriptors (**241**) copied (**237**) from predefined model completion descriptors (**252**) at positions in an array (**264**) of predefined model completion descriptors that correspond to slots in each completion descriptor (**241**) injected into the injection FIFO buffer (**218**).

**[0070]** In the method of FIG. **6**, however, the AMI includes an array (**282**) of predefined completion payload bit masks, and processing (**242**) descriptors (**240**, **241**) in the communication adapter includes copying (**268**) a predefined completion payload bit mask (**282**) from a position in an array (**282**) of predefined completion payload bit masks. The position in the array (**282**) of predefined completion payload bit masks from which the predefined completion payload bit mask is copied corresponds to the slot in which a completion descriptor (**241**) is injected in the injection FIFO buffer. Also in the method of FIG. **6**, processing (**242**) descriptors (**240**, **241**) in the communication adapter (**204**) includes combining (**270**) the completion bit mask (**220**) and the copied completion payload bit mask (**282**) with a logical OR operation. Each element in the array (**262**) of predefined completion payload bit masks is preconfigured to correspond to a slot in the injection FIFO buffer by setting a single bit in each predefined completion payload bit mask. Bit **0** is set in the predefined completion payload bit mask in element **0** of the array (**262**) of predefined completion payload bit masks. Bit **1** is set in the predefined completion payload bit mask in element **1** of the array (**262**) of predefined completion payload bit masks. Bit **2** is set in the predefined completion payload bit mask in element **2** of the array (**262**) of predefined completion payload bit masks. And so on through bit N−1 in element N−1.

**[0071]** The logical OR operation can be implemented in a number of ways, including, for example, loading the completion bit mask into a processor register with the copied completion payload bit mask (**282**), executing a logical OR operation on those registers, and storing the result back into the completion bit mask (**220**) wherever it is stored in memory. That is quite a bit of processing, however, and in computers that support a store-OR, the store-OR is probably preferred. The store-OR operation is a single memory operation that both stores into memory and at the same time performs a bitwise logical OR operation with the operand and the contents of the target memory. A store-OR operation on element **0** of the array (**262**) of predefined completion payload bit masks, in which only bit **0** is set, and the completion bit mask (**220**), therefore will precisely set bit **0** in the completion bit mask (**220**), leaving all other bits (**219**) in the completion bit mask (**220**) unchanged. A store-OR operation on element **1** of the array (**262**) of predefined completion payload bit masks, in which only bit **1** is set, and the completion bit mask (**220**), therefore will precisely set bit **1** in the completion bit mask (**220**), leaving all other bits (**219**) in the completion bit mask (**220**) unchanged. A store-OR operation on element **2** of the array (**262**) of predefined completion payload bit masks, in which only bit **2** is set, and the completion bit mask (**220**), therefore will precisely set bit **2** in the completion bit mask (**220**), leaving all other bits (**219**) in the completion bit mask (**220**) unchanged. And so on, through bit N−1.

**[0072]** At this point in explanation, a data processing advantage is seen in using slots and array elements that correspond to a completion descriptor's slot in the injection FIFO buffer. The communications adapter is not required to load and process against the payload address (**254**) in any

completion descriptor (**241**), a substantial data processing overhead. Because the communications adapter already knows the pertinent slot number in the injection FIFO buffer where the corresponding completion descriptor is injected, the communications adapter already knows the pertinent element in the array (**262**) of predefined completion payload bit masks from which to copy (**268**) a completion payload bit mask (**268**) when the adapter (**204**) begins the copying process (**268**). In fact, in at least one embodiment, the payload address (**254**) is omitted entirely from the predefined model completion descriptors (**252**) and therefore also from the completion descriptors (**241**).

**[0073]** Also at this point in explanation, an advantage is seen in the structure of the completion descriptors (**241**) and the predefined model completion descriptors (**252**), including as they do payload address (**254**), payload size (**256**), destination (**258**), and transfer mode (**260**), they are processed in a manner that is very similar, in some embodiments identical, to the processing of transfer descriptors (**240**). The communication adapter (**204**) can treat a completion descriptor (**241**) effectively like a transfer descriptor by treating a copied predefined completion payload bit mask (**282**) just like any other payload, retrieve it in its specified size (**256**) from memory and transfer it to its destination (**258**) just like another data communications payload. The only thing that distinguishes it is the fact that its destination is always a completion bit mask (**220**) and its delivery typically includes a logical OR. A communications adapter (**204**) that already supports destination addresses in memory and mode specifications can be 'adapted' for completion processing for data communications instructions according to embodiments of the present invention, therefore, with no need for any modifications to the adapter.

**[0074]** For further explanation, FIG. **7** sets forth a flow chart illustrating a further example method of completion processing for data communications instructions in a distributed computing environment according to embodiments of the present invention. The method of FIG. **7** is similar to the method of FIG. **6** in that it also is implemented in a distributed computing environment like that described above with reference to FIG. **1**, a distributed computing environment that includes a plurality of computers coupled for data communications through communications adapters and through an AMI. The method of FIG. **7** is also like the method of FIG. **6** in that it includes injecting (**236**) transfer descriptors (**240**); injecting (**237**) completion descriptors (**241**) for instructions that specify a callback (**223**); listing (**238**) in a pending callback list (**210**) any callback function (**223**) specified by an instruction; processing (**242**) descriptors (**240**, **241**) through the communications adapter (**204**), setting a bit (**221**) in a completion bit mask (**220**) for each completion descriptor processed; and calling (**248**) any callback functions in the pending callback list (**210**) for which transfers of payload data have been completed as indicated by set bits in the completion bit mask (**220**). Also in the method of FIG. **7**, like the method of FIG. **6**, the AMI injects (**237**) completion descriptors (**241**) copied (**237**) from predefined model completion descriptors (**252**) at positions in an array (**264**) of predefined model completion descriptors that correspond to slots in each completion descriptor (**241**) injected into the injection FIFO buffer (**218**). Further the method of FIG. **7** is similar to the method of FIG. **6** in that processing (**242**) descriptors is carried out by copying a predefined completion payload bit mask (**282**) from a position in an array (**262**) of predefined completion payload bit masks corresponding to a completion descriptor's slot in the injection FIFO buffer (**218**) and combining (**270**) the completion bit mask (**220**) and the copied completion payload bit mask (**282**) with a logical OR operation.

**[0075]** In the method of FIG. **7**, however, listing (**238**) the callback function (**223**) includes incrementing (**239**) a pending callback counter (**216**) for each listed callback function. Also in the method of FIG. **7**, callback functions in the pending callback list (**210**) are called (**248**) only when the pending callback counter value is greater than zero (**272**, **274**). In addition, when callbacks are called (**248**), calling (**248**) callbacks includes decrementing (**239**) the pending callback counter (**216**) for each callback function called. Thus the pending callback counter always registers the number of pending callback functions (**223**) presently listed in the pending callback list (**210**), and, in any given iteration through the functions of the AMI, if the value registered by the pending callback counter is zero (**272**, **276**), the AMI skips the step of calling (**248**) callbacks, because there are no callbacks listed in the pending callback list (**210**), and continues (**250**) to a next iteration through the functions of the AMI. Without the counter (**216**), the AMI has no way of knowing in any particular iteration, whether there are any callbacks listed in the pending callback list (**210**), which means that, without the counter, the AMI must bitwise scan the entire completion bit mask (**220**) to determine that there are no callbacks listed in the pending callback list—and in embodiments, the completion bit mask (**220**) contains a thousand bits or more. Saving thousands of bitwise comparisons represents a substantial improvement in data processing efficiency, and that is the purpose of the pending callback counter (**216**). Readers also will recognize that the determination (**272**) whether the pending callback counter registers zero is effectively a determination determining whether the pending callback list (**210**) presently lists callback functions for which transfers of payload data have been completed and that this determination is implemented with no storage, sorting, processing or use whatsoever of any message sequence numbers.

**[0076]** Example embodiments of the present invention are described largely in the context of a fully functional computer system for completion processing for data communications instructions in a distributed computing environment. Readers of skill in the art will recognize, however, that the present invention also may be embodied in a computer program product disposed upon computer readable storage media for use with any suitable data processing system. Such computer readable storage media may be any storage medium for machine-readable information, including magnetic media, optical media, or other suitable media. Examples of such media include magnetic disks in hard drives or diskettes, compact disks for optical drives, magnetic tape, and others as will occur to those of skill in the art. Persons skilled in the art will immediately recognize that any computer system having suitable programming means will be capable of executing the steps of the method of the invention as embodied in a computer program product. Persons skilled in the art will recognize also that, although some of the example embodiments described in this specification are oriented to software installed and executing on computer hardware, nevertheless, alternative embodiments implemented as firmware or as hardware are well within the scope of the present invention.

**[0077]** Example embodiments of the present invention are described largely in the context of fully functional computers

that implements completion processing for data communications instructions in a distributed computing environment according to embodiments of the present invention. Readers of skill in the art will recognize, however, that the present invention also may be embodied in a computer program product disposed upon computer readable storage media for use with any suitable data processing system. Such computer readable storage media may be any storage medium for machine-readable information, including magnetic media, optical media, or other suitable media. Examples of such media include magnetic disks in hard drives or diskettes, compact disks for optical drives, magnetic tape, and others as will occur to those of skill in the art. Persons skilled in the art will immediately recognize that any computer system having suitable programming means will be capable of executing the steps of the method of the invention as embodied in a computer program product. Persons skilled in the art will recognize also that, although some of the example embodiments described in this specification are oriented to software installed and executing on computer hardware, nevertheless, alternative embodiments implemented as firmware or as hardware are well within the scope of the present invention.

**[0078]** As will be appreciated by those of skill in the art, aspects of the present invention may be embodied as method, apparatus or system, or computer program product. Accordingly, aspects of the present invention may take the form of an entirely hardware embodiment or an embodiment combining software and hardware aspects (firmware, resident software, micro-code, microcontroller-embedded code, and the like) that may all generally be referred to herein as a "circuit," "module," "system," or "apparatus." Furthermore, aspects of the present invention may take the form of a computer program product embodied in one or more computer readable media having computer readable program code embodied thereon.

**[0079]** Any combination of one or more computer readable media may be utilized. Such a computer readable medium may be a computer readable signal medium or a computer readable storage medium. A computer readable storage medium may be, for example, but not limited to, an electronic, magnetic, optical, electromagnetic, infrared, or semiconductor system, apparatus, or device, or any suitable combination of the foregoing. More specific examples (a non-exhaustive list) of the computer readable storage medium would include the following: an electrical connection having one or more wires, a portable computer diskette, a hard disk, a random access memory (RAM), a read-only memory (ROM), an erasable programmable read-only memory (EPROM or Flash memory), an optical fiber, a portable compact disc read-only memory (CD-ROM), an optical storage device, a magnetic storage device, or any suitable combination of the foregoing. In the context of this document, a computer readable storage medium may be any tangible medium that can contain, or store a program for use by or in connection with an instruction execution system, apparatus, or device.

**[0080]** A computer readable signal medium may include a propagated data signal with computer readable program code embodied therein, for example, in baseband or as part of a carrier wave. Such a propagated signal may take any of a variety of forms, including, but not limited to, electro-magnetic, optical, or any suitable combination thereof. A computer readable signal medium may be any computer readable medium that is not a computer readable storage medium and that can communicate, propagate, or transport a program for use by or in connection with an instruction execution system, apparatus, or device. Program code embodied on a computer readable medium may be transmitted using any appropriate medium, including but not limited to wireless, wireline, optical fiber cable, RF, etc., or any suitable combination of the foregoing.

**[0081]** Computer program code for carrying out operations for aspects of the present invention may be written in any combination of one or more programming languages, including an object oriented programming language such as Java, Smalltalk, C++ or the like and conventional procedural programming languages, such as the "C" programming language or similar programming languages. The program code may execute entirely on the user's computer, partly on the user's computer, as a stand-alone software package, partly on the user's computer and partly on a remote computer or entirely on the remote computer or server. In the latter scenario, the remote computer may be connected to the user's computer through any type of network, including a local area network (LAN) or a wide area network (WAN), or the connection may be made to an external computer (for example, through the Internet using an Internet Service Provider).

**[0082]** Aspects of the present invention are described in this specification with reference to flowchart illustrations and/or block diagrams of methods, apparatus (systems) and computer program products according to embodiments of the invention. It will be understood that each block of the flowchart illustrations and/or block diagrams, and combinations of blocks in the flowchart illustrations and/or block diagrams, can be implemented by computer program instructions. These computer program instructions may be provided to a processor of a general purpose computer, special purpose computer, or other programmable data processing apparatus to produce a machine, such that the instructions, which execute via the processor of the computer or other programmable data processing apparatus, create means for implementing the functions/acts specified in the flowchart and/or block diagram block or blocks.

**[0083]** These computer program instructions may also be stored in a computer readable medium that can direct a computer, other programmable data processing apparatus, or other devices to function in a particular manner, such that the instructions stored in the computer readable medium produce an article of manufacture including instructions which implement the function/act specified in the flowchart and/or block diagram block or blocks.

**[0084]** The computer program instructions may also be loaded onto a computer, other programmable data processing apparatus, or other devices to cause a series of operational steps to be performed on the computer, other programmable apparatus or other devices to produce a computer implemented process such that the instructions which execute on the computer or other programmable apparatus provide processes for implementing the functions/acts specified in the flowchart and/or block diagram block or blocks.

**[0085]** The flowcharts and block diagrams in the Figures illustrate the architecture, functionality, and operation of possible implementations of computer apparatus, methods, and computer program products according to various embodiments of the present invention. In this regard, each block in a flowchart or block diagram may represent a module, segment, or portion of code, which comprises one or more executable instructions for implementing the specified logical function

(s). It should also be noted that, in some alternative implementations, the functions noted in the block may occur out of the order noted in the figures. For example, two blocks shown in succession may, in fact, be executed substantially concurrently, or the blocks may sometimes be executed in the reverse order, depending upon the functionality involved. It will also be noted that each block of the block diagrams and/or flowchart illustrations, and combinations of blocks in the block diagrams and/or flowchart illustration, can be implemented by special purpose hardware-based systems that perform the specified functions or acts, or combinations of special purpose hardware and computer instructions.

**[0086]** It will be understood from the foregoing description that modifications and changes may be made in various embodiments of the present invention without departing from its true spirit. The descriptions in this specification are for purposes of illustration only and are not to be construed in a limiting sense. The scope of the present invention is limited only by the language of the following claims.

What is claimed is:

**1**. A method of completion processing for data communications instructions in a distributed computing environment, the distributed computing environment including a plurality of computers coupled for data communications through communications adapters and an active messaging interface ('AMI'), the method comprising:

injecting, by the AMI for each of a sequence of data communications instructions into a slot in an injection FIFO buffer of a data communication adapter, a transfer descriptor specifying to the communications adapter a transfer of payload data according to each data communications instruction, at least some of the instructions specifying callback functions;

injecting by the AMI a completion descriptor for each instruction that specifies a callback function into the next slot after that instruction's transfer descriptor in the injection FIFO buffer, the slot in which the completion descriptor is injected having a corresponding slot in a pending callback list;

listing, by the AMI in the corresponding slot in the pending callback list for each data communications instruction that specifies a callback function, the callback function specified by that instruction;

processing by the communications adapter each descriptor in the injection FIFO buffer, including transferring payload data as specified by each transfer descriptor and setting, as payload data for each completion descriptor, a bit that corresponds in a completion bit mask to the slot in the FIFO where the completion descriptor was injected, the completion bit mask comprising a plurality of bits, each bit corresponding to a slot in the injection FIFO buffer; and

calling by the AMI any callback functions in the pending callback list for which transfers of payload data have been completed as indicated by set bits in the completion bit mask.

**2**. The method of claim **1** further comprising determining by the AMI for each data communications instruction that space is available in the injection FIFO buffer of the communications adapter for that instruction's injection descriptor, including determining that a slot is open in the injection FIFO buffer and that a corresponding slot contains a null value in the pending callback list.

**3**. The method of claim **1** wherein calling callback functions further comprises calling each callback function whose position in the pending callback list corresponds to a set bit in the completion bit mask, including resetting each such bit after calling the corresponding callback function.

**4**. The method of claim **1** wherein injecting a completion descriptor further comprises:

copying a predefined model completion descriptor from a position in an array of predefined model completion descriptors that corresponds to the slot in which the completion descriptor is injected in the injection FIFO buffer, the array of predefined model completion descriptors comprising predefined model completion descriptors for all slots in the injection FIFO buffer.

**5**. The method of claim **1** wherein setting a bit that corresponds in a completion bit mask to the slot in the FIFO where the completion descriptor was injected further comprises:

copying a predefined completion payload bit mask from a position in an array of predefined completion payload bit masks that corresponds to the slot in which the completion descriptor is injected in the injection FIFO buffer, the array of predefined completion payload bit masks comprising predefined completion payload bit masks corresponding to each slot in the injection FIFO buffer.

**6**. The method of claim **1** wherein setting a bit that corresponds in a completion bit mask to the slot in the FIFO where the completion descriptor was injected further comprises:

copying a predefined completion payload bit mask from a position in an array of predefined completion payload bit masks that corresponds to the slot in which the completion descriptor is injected in the injection FIFO buffer; and

combining the completion bit mask and the copied completion payload bit mask with a logical OR operation.

**7**. The method of claim **1** wherein each completion descriptor specifies:

as payload data for a data transfer a predefined completion payload bit mask in an array of predefined completion payload bit masks;

the completion bit mask as a target of a data transfer; and

a store-OR memory operation as a mode of a data transfer.

**8**. The method of claim **1** wherein:

listing the callback function further comprises incrementing a pending callback counter for each listed callback function; and

calling callback functions further comprises calling callback functions only when the pending callback counter value is greater than zero and decrementing the pending callback counter for each callback function called.

**9**. Apparatus for completion processing for data communications instructions in a distributed computing environment, the apparatus comprising a plurality of computers disposed within the distributed computing environment with the computers coupled for data communications through communications adapters and an active messaging interface ('AMI'), the computers comprising computer processors operatively coupled to computer memory having disposed within it computer program instructions that, when executed by the computer processors, cause the apparatus to function by:

injecting, by the AMI for each of a sequence of data communications instructions into a slot in an injection FIFO buffer of a data communication adapter, a transfer descriptor specifying to the communications adapter a

transfer of payload data according to each data communications instruction, at least some of the instructions specifying callback functions;

injecting by the AMI a completion descriptor for each instruction that specifies a callback function into the next slot after that instruction's transfer descriptor in the injection FIFO buffer, the slot in which the completion descriptor is injected having a corresponding slot in a pending callback list;

listing, by the AMI in the corresponding slot in the pending callback list for each data communications instruction that specifies a callback function, the callback function specified by that instruction;

processing by the communications adapter each descriptor in the injection FIFO buffer, including transferring payload data as specified by each transfer descriptor and setting, as payload data for each completion descriptor, a bit that corresponds in a completion bit mask to the slot in the FIFO where the completion descriptor was injected, the completion bit mask comprising a plurality of bits, each bit corresponding to a slot in the injection FIFO buffer; and

calling by the AMI any callback functions in the pending callback list for which transfers of payload data have been completed as indicated by set bits in the completion bit mask.

**10**. The apparatus of claim **9** further comprising computer program instructions that, when executed by the computer processors, cause the apparatus to function by determining by the AMI for each data communications instruction that space is available in the injection FIFO buffer of the communications adapter for that instruction's injection descriptor, including determining that a slot is open in the injection FIFO buffer and that a corresponding slot contains a null value in the pending callback list.

**11**. The apparatus of claim **9** wherein calling callback functions further comprises calling each callback function whose position in the pending callback list corresponds to a set bit in the completion bit mask, including resetting each such bit after calling the corresponding callback function.

**12**. The apparatus of claim **9** wherein injecting a completion descriptor further comprises copying a predefined model completion descriptor from a position in an array of predefined model completion descriptors that corresponds to the slot in which the completion descriptor is injected in the injection FIFO buffer, the array of predefined model completion descriptors comprising predefined model completion descriptors for all slots in the injection FIFO buffer.

**13**. The apparatus of claim **9** wherein setting a bit that corresponds in a completion bit mask to the slot in the FIFO where the completion descriptor was injected further comprises copying a predefined completion payload bit mask from a position in an array of predefined completion payload bit masks that corresponds to the slot in which the completion descriptor is injected in the injection FIFO buffer, the array of predefined completion payload bit masks comprising predefined completion payload bit masks corresponding to each slot in the injection FIFO buffer.

**14**. The apparatus of claim **9** wherein:

listing the callback function further comprises incrementing a pending callback counter for each listed callback function; and

calling callback functions further comprises calling callback functions only when the pending callback counter value is greater than zero and decrementing the pending callback counter for each callback function called.

**15**. A computer program product for completion processing for data communications instructions in a distributed computing environment, the distributed computing environment including a plurality of computers coupled for data communications through communications adapters and an active messaging interface ('AMI'), the computer program product comprising computer program instructions that, when installed and executed, cause the parallel computer to function by:

injecting, by the AMI for each of a sequence of data communications instructions into a slot in an injection FIFO buffer of a data communication adapter, a transfer descriptor specifying to the communications adapter a transfer of payload data according to each data communications instruction, at least some of the instructions specifying callback functions;

injecting by the AMI a completion descriptor for each instruction that specifies a callback function into the next slot after that instruction's transfer descriptor in the injection FIFO buffer, the slot in which the completion descriptor is injected having a corresponding slot in a pending callback list;

listing, by the AMI in the corresponding slot in the pending callback list for each data communications instruction that specifies a callback function, the callback function specified by that instruction;

processing by the communications adapter each descriptor in the injection FIFO buffer, including transferring payload data as specified by each transfer descriptor and setting, as payload data for each completion descriptor, a bit that corresponds in a completion bit mask to the slot in the FIFO where the completion descriptor was injected, the completion bit mask comprising a plurality of bits, each bit corresponding to a slot in the injection FIFO buffer; and

calling by the AMI any callback functions in the pending callback list for which transfers of payload data have been completed as indicated by set bits in the completion bit mask.

**16**. The computer program product of claim **15** further comprising computer program instructions that, when installed and executed, cause the parallel computer to function by determining by the AMI for each data communications instruction that space is available in the injection FIFO buffer of the communications adapter for that instruction's injection descriptor, including determining that a slot is open in the injection FIFO buffer and that a corresponding slot contains a null value in the pending callback list.

**17**. The computer program product of claim **15** wherein calling callback functions further comprises calling each callback function whose position in the pending callback list corresponds to a set bit in the completion bit mask, including resetting each such bit after calling the corresponding callback function.

**18**. The computer program product of claim **15** wherein injecting a completion descriptor further comprises copying a predefined model completion descriptor from a position in an array of predefined model completion descriptors that corresponds to the slot in which the completion descriptor is injected in the injection FIFO buffer, the array of predefined

model completion descriptors comprising predefined model completion descriptors for all slots in the injection FIFO buffer.

**19**. The computer program product of claim **15** wherein setting a bit that corresponds in a completion bit mask to the slot in the FIFO where the completion descriptor was injected further comprises:

copying a predefined completion payload bit mask from a position in an array of predefined completion payload bit masks that corresponds to the slot in which the completion descriptor is injected in the injection FIFO buffer; and

combining the completion bit mask and the copied completion payload bit mask with a logical OR operation.

**20**. The computer program product of claim **15** wherein:

listing the callback function further comprises incrementing a pending callback counter for each listed callback function; and

calling callback functions further comprises calling callback functions only when the pending callback counter value is greater than zero and decrementing the pending callback counter for each callback function called.

* * * * *